

# BC402

## Advanced ABAP

SAP NetWeaver

Date \_\_\_\_\_  
Training Center \_\_\_\_\_  
Instructors \_\_\_\_\_  
Education Website \_\_\_\_\_

### Participant Handbook

Course Version: 92  
Course Duration: 5 Day(s)  
Material Number: 50099912



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

## Copyright

Copyright © 2011 SAP AG. All rights reserved.

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

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

## Trademarks

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

## Disclaimer

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

# About This Handbook

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

## Typographic Conventions

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

Type Style	Description
<i>Example text</i>	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal and external.
<b>Example text</b>	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
Example text	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
<b>Example text</b>	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

## Icons in Body Text

The following icons are used in this handbook.

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

# Contents

<b>Course Overview .....</b>	<b>vii</b>
Course Goals .....	vii
Course Objectives .....	vii
<b>Unit 1: ABAP Basics .....</b>	<b>1</b>
Program Types and Modularization .....	2
Data Types and Data Objects .....	12
Database Access with Open SQL .....	24
<b>Unit 2: Program Calls and Memory Management .....</b>	<b>35</b>
Calling Programs .....	37
ABAP Runtime and Memory Management .....	54
Shared Objects .....	94
<b>Unit 3: Processing Internal Data .....</b>	<b>125</b>
Arithmetic Expressions .....	127
Processing Byte Strings and Character Strings .....	146
Table Types and Their Use .....	189
Special Techniques for Using Internal Tables .....	213
Using Data References and Field Symbols .....	235
<b>Unit 4: Dynamic Programming .....</b>	<b>253</b>
Overview .....	255
Dynamic Calls and Program Generation .....	258
Generic Data Types and Dynamic Access to Data Objects .....	275
Querying Type Attributes at Runtime .....	304
Generating Objects, Data Objects, and Data Types at Runtime .....	329
<b>Unit 5: ABAP Open SQL in Detail .....</b>	<b>361</b>
Database, Database Interface, and Open SQL .....	363
WHERE Condition and Target Area .....	380
SELECT Statements: Processing and Aggregating Value Sets .....	390
Reading from Multiple Database Tables .....	409
Streams and Locators .....	448
<b>Unit 6: Analysis and Testing .....</b>	<b>459</b>
Breakpoints, Assertions, and Logpoints .....	460

<b>Unit 7: Hints and Tips – Proven Techniques .....</b>	<b>479</b>
Frequent Mistakes and How to Avoid Them .....	480
Recommendations and Conventions.....	486
Performance (ABAP).....	491
Performance (Database Access) .....	500
<b>Appendix 1: ABAP Debugger .....</b>	<b>509</b>
<b>Appendix 2: Runtime Analysis .....</b>	<b>519</b>
<b>Appendix 3: The Code Inspector .....</b>	<b>525</b>
<b>Appendix 4: SQL Trace .....</b>	<b>531</b>
<b>Index .....</b>	<b>539</b>

# Course Overview

## Target Audience

This course is intended for the following audiences:

- Developers
- Consultants

## Course Prerequisites

### Required Knowledge

- SAPTEC (Fundamentals of SAP NetWeaver Application Server)
- BC400 (ABAP Workbench – Foundation)
- BC401 (ABAP Objects)

### Recommended Knowledge

- Practical programming experience with ABAP (Objects)
- BC430 (ABAP Dictionary)



## Course Goals

This course will prepare you to:

- Enhance your skills with the ABAP programming language
- Develop complex, high-performance business applications with ABAP



## Course Objectives

After completing this course, you will be able to:

- Explain how the ABAP runtime environment works
- Use complex ABAP statements and their variants
- Develop complex ABAP applications with dynamic components
- Use advanced techniques with Open SQL for database access
- Create, test, compare, and classify ABAP applications

## SAP Software Component Information

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

- SAP NetWeaver 7.0EhP 2

# Unit 1

## ABAP Basics

### Unit Overview

This unit summarizes the major techniques and knowledge that you should be familiar with from the previous courses, BC400 and BC401. This includes ABAP program types, modularization techniques, and an understanding of the data types and data objects used in ABAP. You should also be familiar with the syntax of database read accesses with Open SQL.



### Unit Objectives

After completing this unit, you will be able to:

- Explain the basic structures of ABAP programs
- Name the different types of ABAP programs
- Describe the components of the various ABAP program types
- Use the different modularization techniques available in ABAP
- Describe the type system in ABAP
- Name the respective advantages of global and local types
- Differentiate between flat and deep data objects
- Explain the visibility and validity of data objects
- Explain what Open SQL is
- Understand and program simple database read accesses with Open SQL
- Explain the difference between single record access, sequential access, and array fetch

### Unit Contents

Lesson: Program Types and Modularization .....	2
Exercise 1: Creating Packages .....	9
Lesson: Data Types and Data Objects.....	12
Lesson: Database Access with Open SQL.....	24
Exercise 2: Creating an Executable Program .....	29

# Lesson: Program Types and Modularization

## Lesson Overview

This lesson describes the basic structure and components of ABAP programs. A particular focus lies on the differences between the different program types.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basic structures of ABAP programs
- Name the different types of ABAP programs
- Describe the components of the various ABAP program types
- Use the different modularization techniques available in ABAP

## Business Example

You want to develop complex applications that are highly modularized. To be sure you choose the right modularization technique, get a general overview of the structure, components, and functions of the different ABAP program types.

## Structure of an ABAP Program

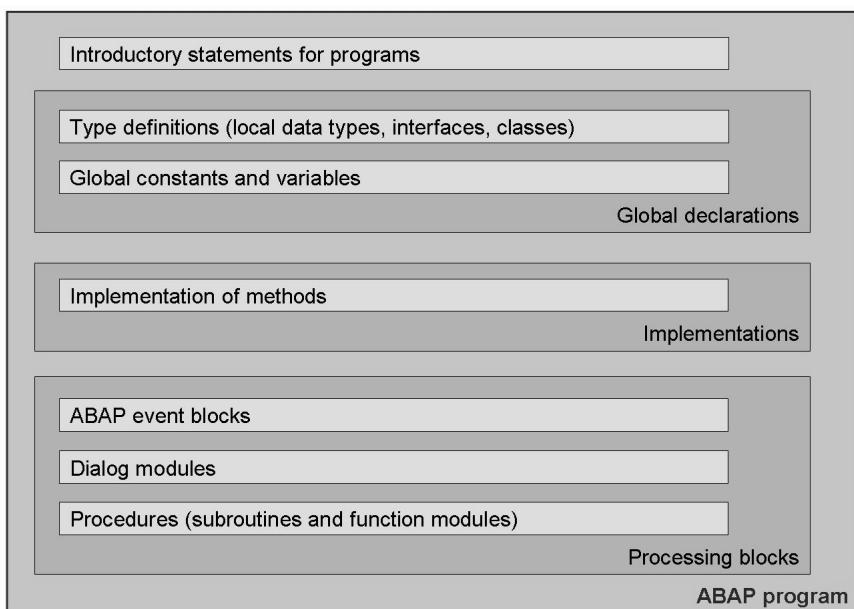


Figure 1: Structure of the ABAP Source Code

The source code of an ABAP program consists of several parts. Which of these parts is actually contained in a given program depends on the respective program type.

### Introductory statements for programs

Nearly every program type has its own special introductory statement. Every program can only have one introductory statement. It must be the **first statement** in the program (after expanding any includes).

### Global declarations

Include the declarative statements global program data objects (constants and variables), as well as for local program data types, classes, and interfaces.

### Implementations

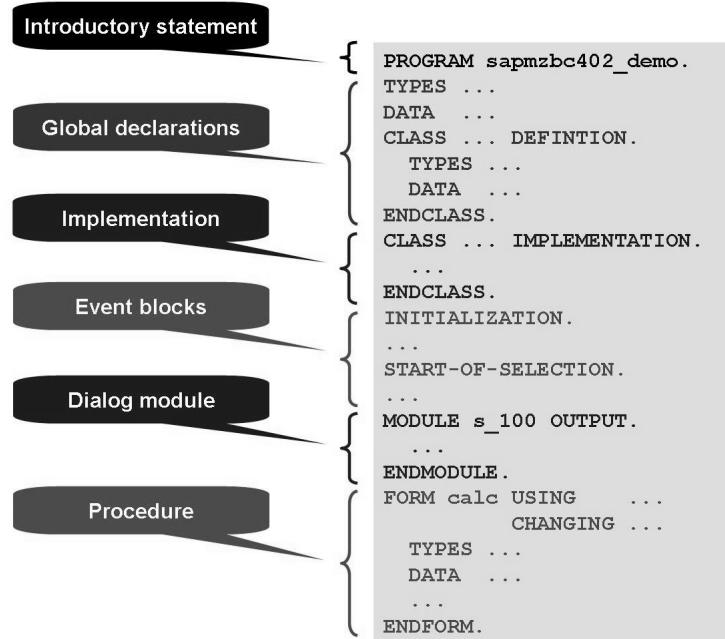
Include the implementation parts of the local classes, with the method implementations of these classes.

### Processing blocks

A processing block is an indivisible program unit that must be programmed contiguously. There are three types of processing blocks:

- **ABAP event blocks** are introduced by an event keyword and ended implicitly by keywords that introduce the next processing block. Event blocks are called by the ABAP runtime environment at specific times during program execution. They cannot be called in the source code.
- **Dialog modules** are introduced and concluded using the relevant keywords (MODULE ... ENDMODULE). Dialog modules can only be addressed in the flow logic of a screen. They are then called by the runtime environment when the screen is processed.
- **Procedures** are also introduced and concluded using the relevant keywords (FORM ... ENDFORM, and FUNCTION ... ENDFUNCTION). They are reusable units that you use to modularize the program. They are executed when they are explicitly called in a different processing block.

The following diagram shows the structure of the source code, using a module pool as an example.



**Figure 2: Structure of the ABAP Source Code**

## Introductory Statements for Programs

The introductory statement for a program depends on the respective program type, specifically:



### Introductory Statements for Programs

Program type	Introductory statement
Executable program	REPORT name ... [MESSAGE-ID id].
Function group	FUNCTION-POOL name ... [MESSAGE-ID id].
Module pool	PROGRAM name [MESSAGE-ID id].
Subroutine pool	No separate statement
Class pool (global class)	CLASS-POOL [MESSAGE-ID id].
Interface pool (global interface)	INTERFACE-POOL.
Type pool (Type group)	TYPE-POOL name.

When developers create new repository objects, they define a program type, which then appears in the program attributes. When the program is created, the development environment generates the suitable introductory statement in the source code automatically.

For some program types, the developer can add the optional addition MESSAGE-ID later, to specify a standard message class. Additions for controlling list output are also available for the REPORT and FUNCTION-POOL statements.



**Caution:** The introductory statement does not necessarily have to match the program type in the program attributes.

To avoid unexpected system behavior, when editing, make sure you only change the additions, but not the statement itself.

## Global Declarations

The diagram below shows, which global declarations are allowed for each program type.



Program type	Local data types	Constants	Global data objects	Local classes, local interfaces
Executable program	✓	✓	✓	✓
Function group	✓	✓	✓	✓
Module pool	✓	✓	✓	✓
Subroutine pool	✓	✓	✓	✓
Class pool (global class)	✓	As attributes	As attributes	✓
Interface pool (global interface)	In interface signature	As attributes	✗	✗
Type pool (type group)	✓	✓	✗	✗

**Figure 3: Global Declarations Dependent on Program Type**

Local program data types can be defined in all program types. For global classes, this is possible in both the class signature (*Types* tab of the *Class Builder*) and in the local definition part of the class (*Local Types* button). Only the first option is available for global interfaces.

Constants cannot be created in the local definition part of a class. They must be defined as part of the signature (*Attributes* tab).

Global interfaces and type groups are collections of data types and constants. Therefore, they cannot contain any other data objects, local classes, or interfaces.

## Processing Blocks

The major difference between the various program types is which processing blocks they can contain.



Program type	ABAP events	Screens and dialog modules	Subroutines	Implementation of methods
Executable program	All	✓	✓	✓
Function group	All (w/o reporting)	✓	✓ (*)	✓
Module pool	All (w/o reporting)	✓	✓	✓
Subroutine pool	✗	✗	✓	✓
Class pool (global class)	✗	✗	✗	✓
Interface pool (global interface)	✗	✗	✗	✗
Type pool (type group)	✗	✗	✗	✗

(\*) Only in function groups with function modules

**Figure 4: Processing Blocks Dependent on Program Type**

The strictly declarative program types, Interface-Pool, and Type-Pool, cannot contain any processing blocks.

Subroutines are not allowed in object-oriented program types or strictly declarative type groups.

Screens, dialog modules, and ABAP event blocks are only allowed in the “conventional” main program types: report program (or simply report), function group, and module pool. ABAP events have constraints with regard to function groups and module pools.

**Executable programs** may contain the events LOAD-OF-PROGRAM, INITIALIZATION, START-OF-SELECTION, GET, or END-OF-SELECTION.

**Function groups and module pools** may only contain the LOAD-OF-PROGRAM event.

Executable programs, function groups and module pools may contain user dialogs of type dynpro, selection screen, or list. If any of these user dialogs is used, the following additional events may be used:

**Dynpros:** PROCESS BEFORE OUTPUT (PBO), PROCESS AFTER INPUT (PAI), PROCESS ON HELP REQUEST (POH), PROCESS On VALUE REQUEST (POV).

**Selection screens:** AT SELECTION-SCREEN OUTPUT, AT SELECTION-SCREEN.

**Lists:** TOP-OF-PAGE, END-OF-PAGE, AT LINE-SELECTION, AT USER-COMMAND, TOP-OF-PAGE DURING LINE-SELECTION.

Function modules, as specific procedures, are only allowed in function groups.

## Program Organization

In the simplest cases, all the source code is entered in one single program. You should use include programs for more complex programs, to make them easier to understand and make it possible to maintain and transport source code components separately.

**Include programs** help you break up ABAP source codes into individual repository objects. An ABAP program can be created with program type "Include Program" in the program attributes. Include programs **cannot have introductory statements** and cannot be generated independently in the ABAP Compiler. Instead, you use INCLUDE statements to integrate include programs with programs that can be generated.

Include programs must contain full statements. An include program can integrate other include programs, but not itself. Include programs do not necessarily have to contain full processing blocks.

The creation of new include programs is supported by the development environment for some program types (report, module pool, subroutine pool), supported partially for others (function group), or completely forced (class pool, interface pool).

We highly recommend following the naming convention for include programs, which is shown in the diagram below based on a function group as an example:

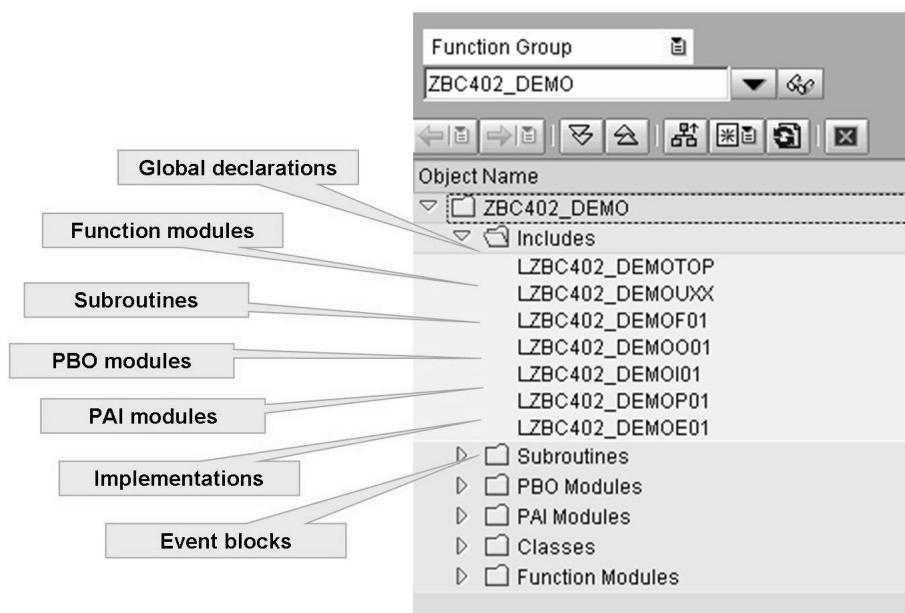


Figure 5: Naming Convention for Includes

**Note:** The INCLUDE statement is the only statement other than an introductory statement that can appear at the top of a program. The only prerequisite is that once the include program is expanded, an introductory statement must appear at the top of the including program.

**Hint:** You can use service report RSINCL00 to generate reference lists for include programs.

# Exercise 1: Creating Packages

## Exercise Objectives

After completing this exercise, you will be able to:

- Create a package
- Assign the repository object to a Workbench request

## Business Example

You have a major development project that is logically self-contained. To structure these developments, create a package that contains all your developments.

### Task:

Create a package in the *ABAP Workbench*.

1. Create a package with name **ZBC402\_##** and assign it to application component **CA**.



**Hint:** ## always stands for your group number.

2. What is the difference between an application component and a software component?

---

---

---

3. What is the importance of the transport layer?

---

---

---

# Solution 1: Creating Packages

## Task:

Create a package in the *ABAP Workbench*.

1. Create a package with name **ZBC402\_##** and assign it to application component **CA**.



**Hint:** ## always stands for your group number.

- a) Start the *Object Navigator*.
  - b) Choose menu path *Workbench* → *Edit Object*. Choose tab page *Development Coordination*.
  - c) Enter a name for the package and choose *Create*.
  - d) Enter a short text.
  - e) Enter the application component.
  - f) Save your entries.
2. What is the difference between an application component and a software component?

**Answer:** The application component is an additional classification criterion for your applications and usually consists of several packages. The application component is used, for example, as a search criterion to find development objects in the Repository Information System.

In turn, a software component contains several application components and is the highest classification criterion in the SAP system. The Basis system, for example, consists of the SAP\_BASIS and SAP\_ABA software components. Your (customer) developments all have to be assigned to software component HOME.

3. What is the importance of the transport layer?

**Answer:** The transport layer determines where your development objects are transported after the Workbench request is released. The system landscape has to be configured accordingly in the *Transport Management System*.



## Lesson Summary

You should now be able to:

- Explain the basic structures of ABAP programs
- Name the different types of ABAP programs
- Describe the components of the various ABAP program types
- Use the different modularization techniques available in ABAP

# Lesson: Data Types and Data Objects

## Lesson Overview

In this lesson, we first read a systematic summary of the type system in ABAP. What types are there? Where and how are they defined and used?

When discussing the data objects, be sure to emphasize the visibility and validity of the data objects. This helps to define the appropriate data objects for the requirements at hand and avoid frequent mistakes.



## Lesson Objectives

After completing this lesson, you will be able to:

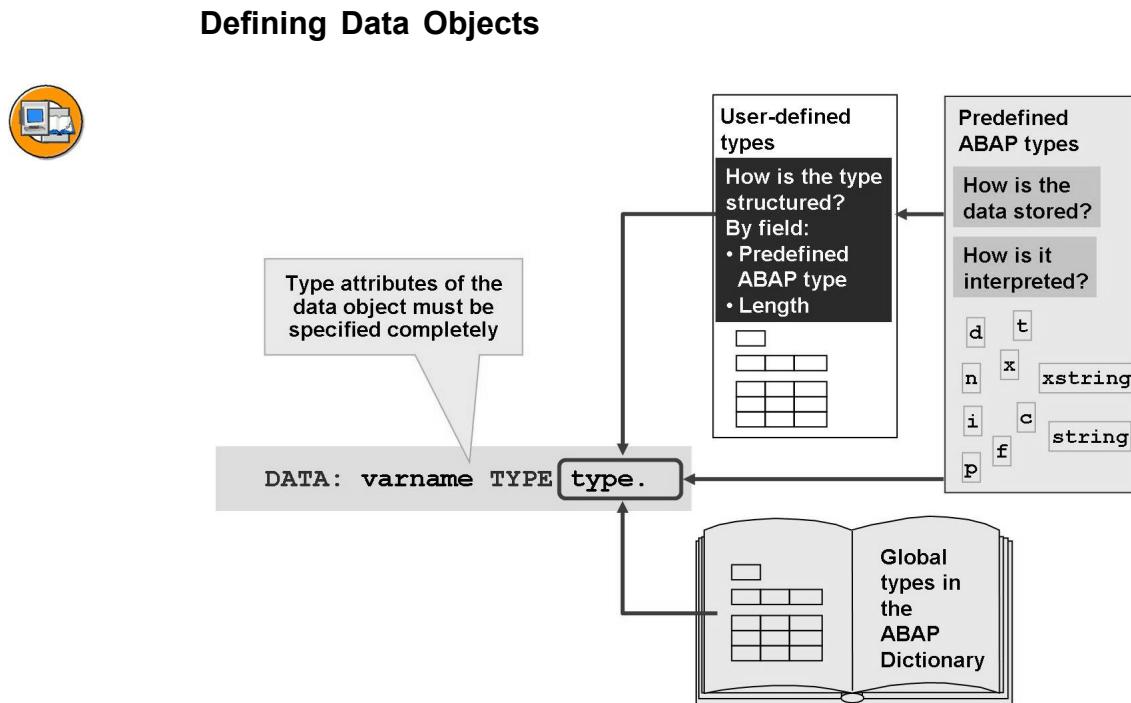
- Describe the type system in ABAP
- Name the respective advantages of global and local types
- Differentiate between flat and deep data objects
- Explain the visibility and validity of data objects

## Business Example

You want to develop sturdy, sophisticated ABAP applications. To do so, gain a systematic overview of the data types and data objects in the ABAP programming language.

## Overview of Data Types and Data Objects

We differentiate between **data objects** and **data types** in ABAP. Data types are mere descriptions that are not linked with an address in the memory. Data objects are instances of data types and occupy memory. They store the specific data that a program uses at runtime.



**Figure 6: Defining Data Objects**

Data objects are defined in a program using the DATA statement. After the name of the data object, a fully specified type is assigned to it using the TYPE addition. The type is linked to the data object statically and cannot be changed at runtime.

Other syntax variants are available (for legacy reasons). Note however, that some of these legacy variants are no longer supported within classes. For more information, see the keyword documentation for the DATA statement.

Predefined data types, global data types, and local data types are available to declare data objects.

### Predefined ABAP types

Predefined data types are provided by the ABAP runtime environment. We differentiate between incomplete types (`c`, `n`, `p`, and `x`) and complete types (`i`, `f`, `d`, `t`, `string`, and `xstring`), depending on whether an additional length specification is required when the corresponding data objects are declared.

### Global data types

Global data types are defined in the Dictionary. They refer to predefined Dictionary types, which largely correspond to the predefined ABAP types.

### Local data types

Local data types are defined using the TYPES statement within the ABAP program. They can refer to predefined ABAP types or global data types from the Dictionary.



## Numeric Data Types

Data Type	Description	Length in Byte / Initial Length	Value Range	Decimals
i	Integer	4	-2,147,483,648 - +2,147,483,647	0
p	Packed Number	1 ... 16 8	Digits: 2*Length – 1 Decimals: up to 14	up to 14
f	Binary Floating Point Number	8	max: +/- 1,79E+308 min: +/- 2,22E-308 and 0	16
decfloat16	Decimal Floating Point Number	8	max: +/- 1E+385 min: +/- 1E-383 and 0	16
decfloat32	Decimal Floating Point Number	16	max: +/- 1E+6145 min: +/- 1E-6143 and 0	32

- DECFLOAT16 and DECFLOAT32 exist as of SAP NW 7.0 EhP2
- Replace type F

- Data types have different:
- Rules for storage
  - Value range
  - Arithmetic used

Figure 7: Predefined ABAP Types - Numerical Types



### Character Data Types

Data Type	Description	Initial Value	Length in Byte / Initial Length	Additions
n	Sequence of digits	0...0	1 ... 65535 1	- Length fixed
c	Sequence of characters	space	1 ... 65535 1	- Length fixed
d	Date Field	00000000 (YYYYMMDD)	8	- Length fixed - Data calculations
t	Time Field	000000 (HHMMSS)	6	- Length fixed - Time calculations
string	Sequence of characters		0 ... 0	- Length variable

Character operations allowed for all types

Figure 8: Predefined ABAP Types - Character Types



### Byte Data Types

Data Type	Description	Initial Value	Length in Byte / Initial Length	Additions
x	Sequence of Bytes in hex. notation	00...	1 ... 65535 1	- Length fixed
xstring	Sequence of Bytes in hex. notation		0 ... 0	- Length variable

Byte operations allowed for both types

Figure 9: Predefined ABAP Types - Byte Types

The above diagrams shows all elementary predefined data types provided by the ABAP runtime environment. The **complete** types (i, f, decfloat16, decfloat32, d, t, string, and xstring) can be used to type a data object directly. The **incomplete** types (p, n, c, and x) must be supplemented with a length specification to create a complete type. You specify the length in the DATA or TYPES statement, usually in brackets after the name of the data object or local data type – for example:

```
TYPE <type_name>(<length>) TYPE <ABAP-Type>.  
DATA <obj_name>(<length>) TYPE <ABAP-Type>.
```



**Hint:** In SAP Web AS 6.10 and later, you can also specify the length after the LENGTH addition – for example:

```
TYPE <type_name> TYPE <ABAP-Type> LENGTH <length>.  
DATA <obj_name> TYPE <ABAP-Type> LENGTH <length>.
```

You can use the DECIMALS addition to define the number of decimal places, especially for type p (packed number).

The **numeric** data types (i, f, p, decfloat16, and decfloat32) differ primarily in their value ranges and in the arithmetic used for calculations. This is described in more detail in a later section.

The major differences between the **character** data types (n, c, d, and t) are the characters that can be used to form the strings of these types and how they are formatted in the screen output. In addition, data objects with type d and t are given special handling in arithmetic operations. We discuss the details further below.

A primary attribute of the **string** data type is that the data objects with this type have **variable lengths**. The runtime system adjusts this length to the respective contents dynamically at runtime.

Each predefined data type has a characteristic initial value. This **type-specific initial value** plays an important role in instantiating the data objects and executing the CLEAR statement. All numeric data types have 0 as their initial value. The character data types n, d, and t have a sequence of character 0 as their initial values, while type c has a sequence of “spaces”. The variable-length data types have a length of zero in their initial state.

## Generic Types

In addition to the predefined ABAP types named above, the runtime environment also provides predefined generic types.



Generic data type	Meaning	Compatible ABAP types
Fully generic any data	Any type Any data type	Any Any
Elementary, flat simple	Elementary data type or flat, character-type structure	Any
Numeric numeric decfloat	Numeric data type Numeric data type	i, f, p, decfloat16, decfloat32 decfloat16, decfloat32
Character-type clike csequence	Character data type Text data type	c, n, d, t, string c, string
Hexadecimal xsequence	Byte data type	x, xstring

Figure 10: Generic ABAP Types

Generic types do not completely define the attributes of a data object, and in contrast to the ABAP types, **cannot** be used to declare data objects. They are used exclusively to type **formal parameters** and **field symbols**.

→ **Note:** Details on the use of generic types are discussed in the context of dynamic programming.

## Data Object Categories



	Reference	Internal table	Structure	Elementary field
Deep	All	All	At least one deep component	string, xstring
Flat	X	X	No deep components	i, f, p, decfloat16, decfloat32, c, n, d, t, x

Figure 11: Data Object Categories

## Data Object Structures

We differentiate between the structures of the following data objects:

- Elementary fields
- Structures
- Internal tables
- References

Structures and internal tables can be mutually nested to nearly any depth. A structure can contain other structures as its components, for example, or even an entire internal table. Such objects are called **nested structures**.

## Flat and Deep Data Objects

With regard to memory requirements and memory management, we also differentiate between **flat** and **deep** data objects:

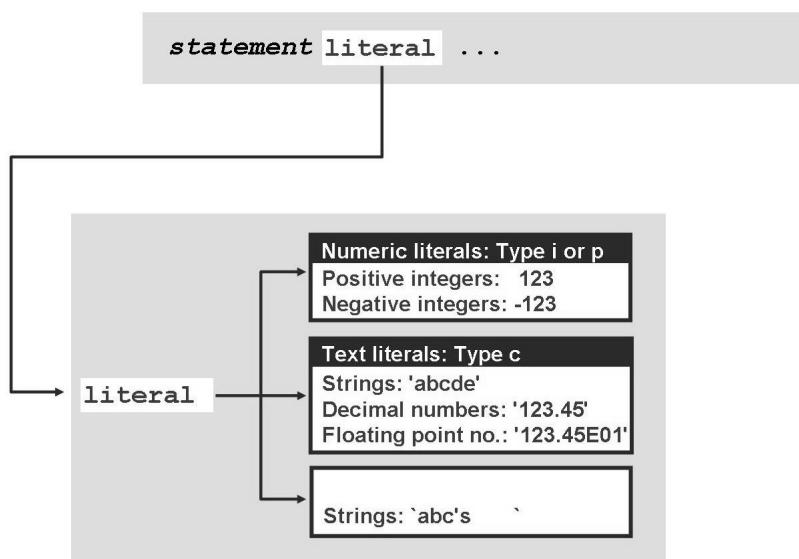
### Flat data objects

Have a static length and do not refer to other objects. All elementary data objects aside from strings and all structures with exclusively flat components (regardless of potential nesting) are flat.

### Deep data objects

References, strings, and internal tables. Structures are also called "deep" if they contain at least one deep component, taking all includes and substructures into account.

## Literals



**Figure 12: Literals**

Literals are data objects that are defined in the source code of a program and whose values are uniquely defined. The ABAP runtime environment differentiates between three types of literals:

#### Numeric literals

You define a numeric literal as a sequence of digits in the program text, with or without +/- signs. A numeric literal has type i if its value lies within the value range of this type (-2<sup>31</sup>+1 to 2<sup>31</sup>-1); otherwise it has type p.

#### Text field literal

You define a text field literal as a character string that you enclose in single quotes (') in the program text. It can be from 1 to 255 characters long. Any spaces at the end of the literal are ignored. Text field literals have type c.

#### String literals

You define a string field literal as a character string that you enclose in back quotes (`) in the program text. String literals have type string. Their length is limited to 255 characters; they can also have 0 length.

In contrast to text field literals, spaces at the end of a string literal are **not** suppressed.

String literals are available in *SAP Web AS 6.10* and later.

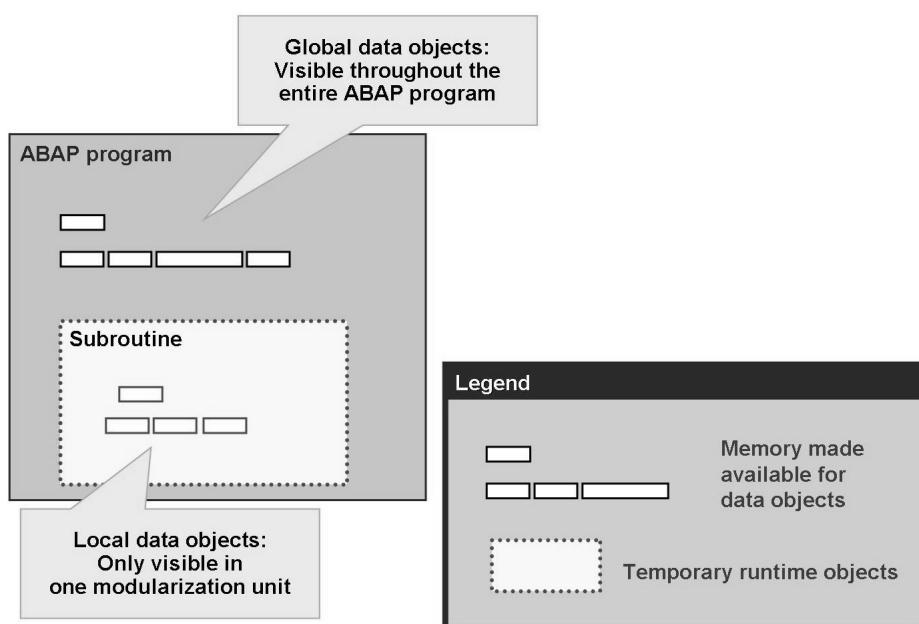


**Hint:** To display a single quote within a text field literal, you have to enter it twice. The same applies to entering a back quote in a string literal. No restrictions apply, however, to single quotes in a string literal or back quotes in a text field literal.



## Visibility and Validity of Data Objects

### Visibility in Conventional ABAP



**Figure 13: Visibility of Data Objects**

The extent to which data objects are visible depends on the context. The following rules apply to data objects that are defined using the DATA statement:

If the DATA statement appears between FORM and ENDFORM, it defines a **local data object of a subroutine**.

If the DATA statement appears between FUNCTION and ENDFUNCTION, it defines a **local data object of a subroutine**.

In all other positions, the DATA statement defines a global data object, which is visible in the entire program.

If a global data object and a local data object have the same name, only the local data object is visible within the modularization unit. In technical terms, it **obscures** the identically named global data object.



#### Caution: Data objects in modules and event blocks are always global.

If the DATA statement appears between MODULE and ENDMODULE, the defined object is still a global data object that is visible in the entire program. Likewise, data objects that you declare in an ABAP event block are visible globally.

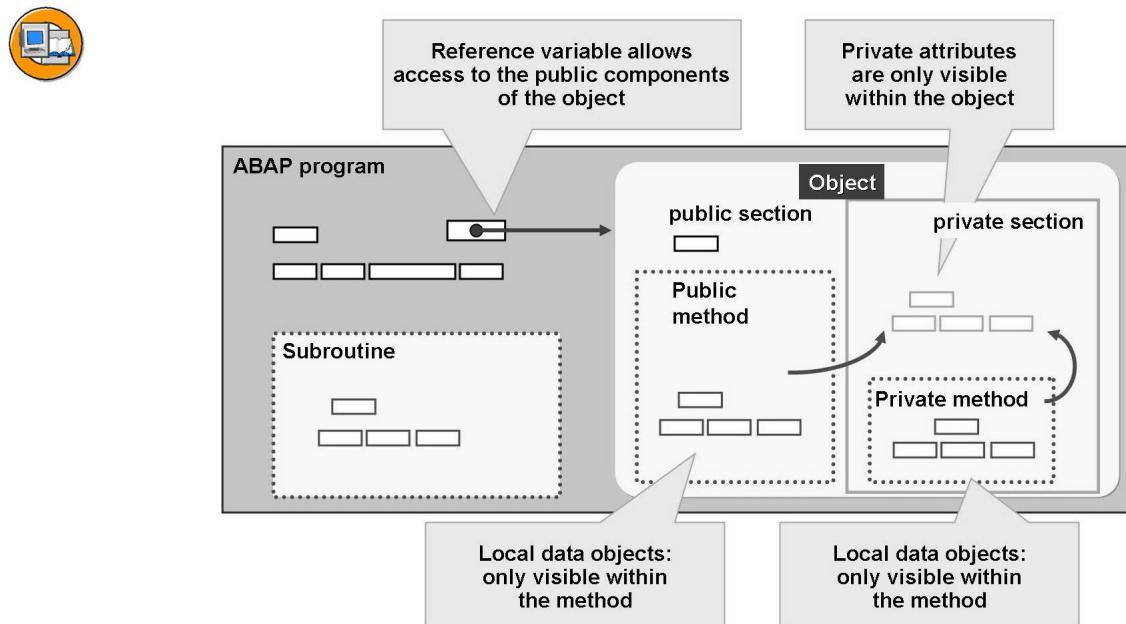
To avoid misunderstandings, we recommend declaring such data at the start of the program and avoiding DATA statements completely in modules and event blocks.

**The TABLES statement always creates data objects that are visible program-wide.**

Data objects that are created with the TABLES statement are always visible in the entire program, even if the statement appears within a subroutine or a function module. (The objects can even be visible in other programs, such as the call of an external subroutine.)

## Visibility of ABAP Objects

If you use objects (that is, instances of ABAP Objects classes) in your application, you have other options for specifying the visibility of your data objects.



**Figure 14: Visibility Areas in Objects**

Data objects that are created in the declaration part of a class are called **attributes**. The following visibility areas exist for attributes:

### Private and protected attributes

These attributes are only visible within the object. All the methods of the object can read and change the data. External access to these attributes is only possible when a suitably defined method is used.

### Public attributes

Public attributes are also visible and accessible outside the object.



### Protected attributes

Protected attributes are visible within the object and its subobjects.

If the DATA statement appears within METHOD .. ENDMETHOD, it defines a local data object in that method.

## Lifetime of Data Objects

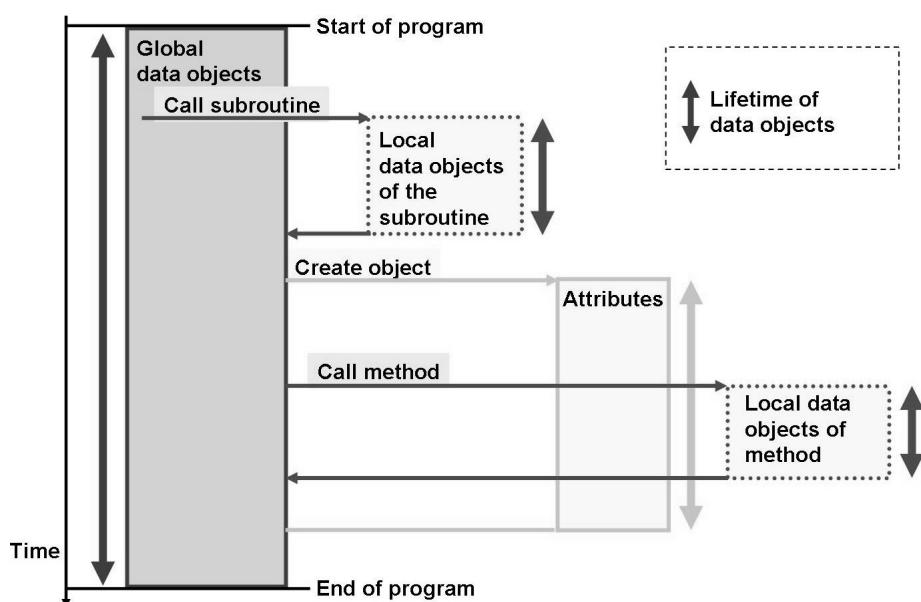


Figure 15: Lifetime of Data Objects

The lifetime of a data object is usually linked with its visibility.

**Global data objects** remain valid as long as the corresponding program is in the memory.

The lifetime of **object attributes** begins when the object is instantiated at runtime and ends when the last reference to the object has been deleted.

The lifetime of a **local data object** is usually identical to the lifetime of the surrounding modularization unit, which means the contents of the data object are lost between two calls of the same modularization unit.



**Hint:** If you use the **STATICS** statement instead of DATA when you declare local data objects, their lifetime is extended to the overall program lifetime. Therefore, such data objects have the same lifetime as global data objects, but their visibility is still limited to the respective modularization unit.

You can use the STATICS statement in function modules, subroutines, and static methods.



## Lesson Summary

You should now be able to:

- Describe the type system in ABAP
- Name the respective advantages of global and local types
- Differentiate between flat and deep data objects
- Explain the visibility and validity of data objects

# Lesson: Database Access with Open SQL

## Lesson Overview

This lesson summarizes what Open SQL is and how simple database read accesses are structured in Open SQL.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain what Open SQL is
- Understand and program simple database read accesses with Open SQL
- Explain the difference between single record access, sequential access, and array fetch

## Business Example

You want to program complex, high-performance database accesses with Open SQL. To do so, in a first step, you refresh your knowledge of the basics of Open SQL.

## Open SQL

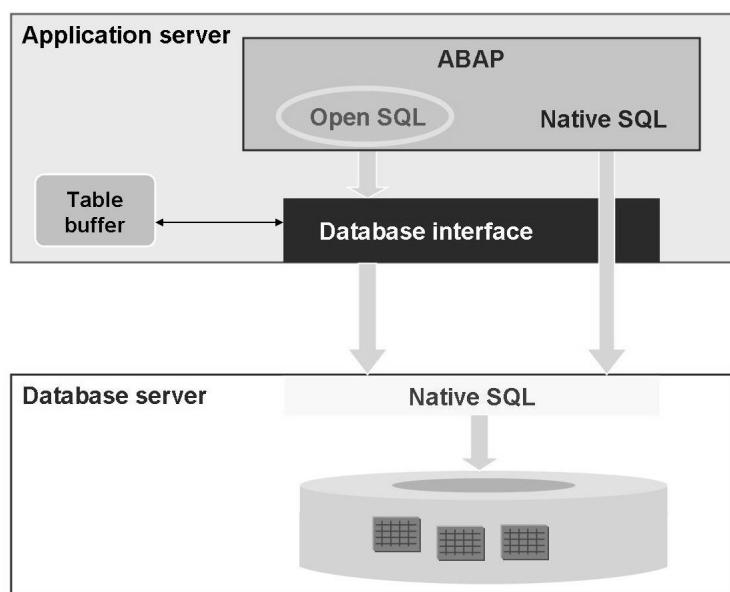
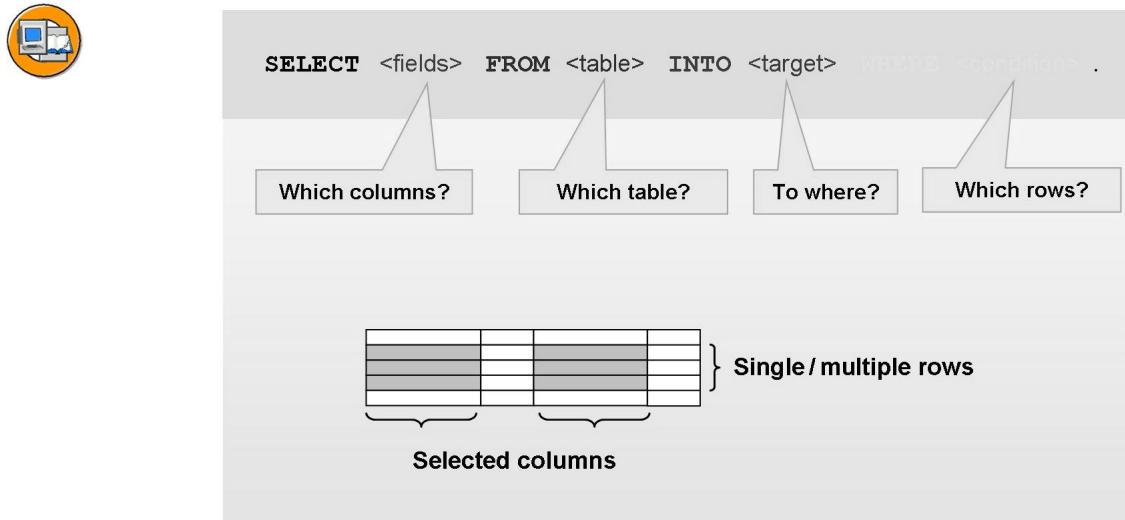


Figure 16: Open SQL and Native SQL

Open SQL statements are a subset of Standard SQL that is fully integrated in the ABAP language. They allow the ABAP programmer uniform access to data, regardless of the database system installed. The database interface dynamically converts Open SQL statements to database-specific SQL statements.

## The SELECT Statement



**Figure 17: Structure of the SELECT Statement**

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

### SELECT clause

The SELECT clause describes, among other things, whether the result of the selection will comprise several lines or a single data record, and which fields of the table are to be read.

### FROM clause

The FROM clause names the source (database table or view) from which the data is to be selected.

### INTO clause

The INTO clause determines the internal data objects into which the selected data is to be placed.

### WHERE clause

The WHERE clause defines the conditions that the selection results must fulfill. It therefore specifies the rows to be selected from the table.

We differentiate between three basic variants of the SELECT statement with regard to the data volume to be read and the structure of the data object to be filled:

**Single record access**

(SELECT SINGLE)

**Sequential processing**

(SELECT ... ENDSELECT)

**ARRAY FETCH**

(SELECT ... INTO TABLE)



**Hint:** Note that the syntax components **SINGLE**, **ENDSELECT**, and **INTO TABLE** are mutually exclusive.



**Note:** However, we will learn about the PACKAGE SIZE addition in a later lesson, which lets you combine INTO TABLE and ENDSELECT in the same SELECT statement.

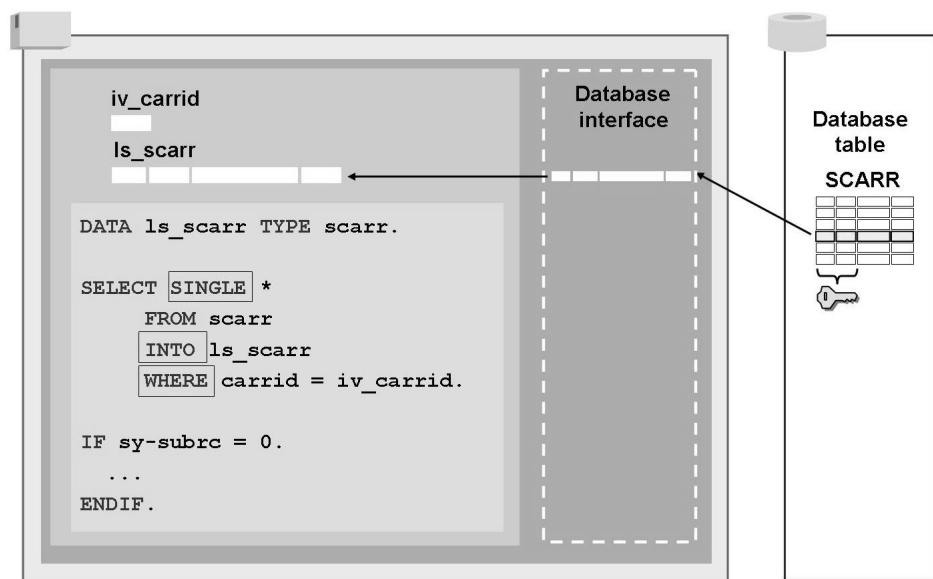
**Single Record Access**

Figure 18: Single Record Access (SELECT SINGLE)

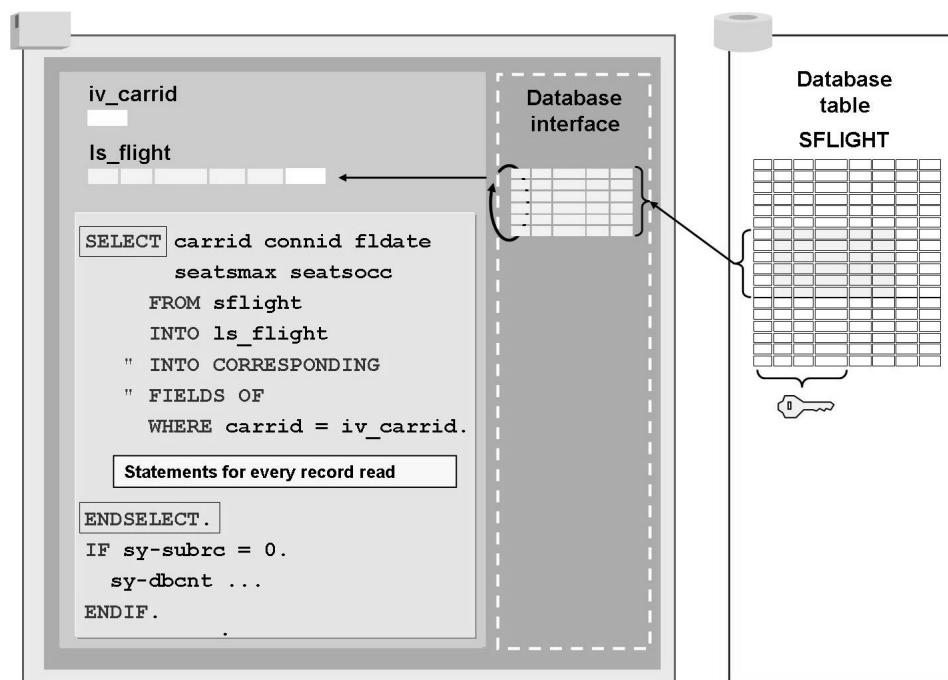
The **SELECT SINGLE \*** statement allows you to read a **single record** from the database table. To ensure unique access, **all key fields** must be filled in the **WHERE clause**.

The **\*** informs the database interface that **all columns** in that line of the database table should be read. If you only wish a specific selection of columns, you can list the required fields instead.

In the **INTO clause**, enter the destination to where the database interface is to copy the data. The target area should have a structure identical to the columns of the database table being read and be **left-justified**. If you use the **CORRESPONDING FIELDS OF** addition to the INTO clause, you can fill the target area component by component. The system only fills those components that have identical names to columns in the database table. If you do not use this addition, the system fills the target area left-justified, irrespective of its structure.

If the system finds a table entry matching your conditions, **sy-subrc** has the value 0.

## Sequential Processing



**Figure 19: Mass Access with Sequential Processing (SELECT ... ENDSELECT)**

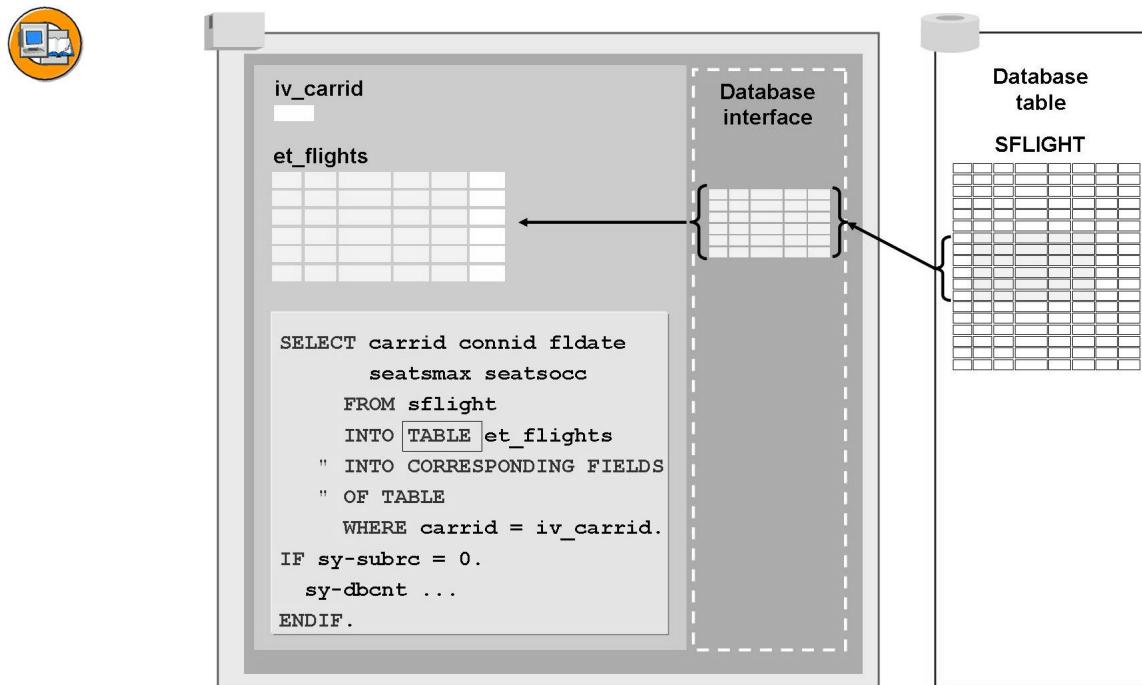
If you do not use the addition **SINGLE** with the **SELECT** statement, the system reads **multiple records** from the database.

The **field list** determines the columns whose data is to be read from the database. The number of requested lines to be read can be restricted using the **WHERE clause**. Within the WHERE clause, you may only enter the field names of the database table. The name of the database table that is accessed is found in the **FROM clause**. Multiple logical conditions can be added to the WHERE clause using AND or OR statements.

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

After the ENDSELECT statement, you can check the return code for the SELECT loop. **sy-subrc** has the value 0 if the solution set consists of at least one record. After the ENDSELECT statement, **sy-dbcnt** contains the total number of lines read.

## ARRAY FETCH



**Figure 20: Mass Access with ARRAY FETCH (SELECT ... INTO TABLE)**

The addition **INTO TABLE ...** addition causes the ABAP runtime system to copy the contents of the database interface directly to the internal table, itab. This is called an **array fetch**. Since the array fetch is not executed as a loop, you **must not program an ENDSELECT statement**.

**sy-subrc** is set to 0 if at least one record was read; in this case, **sy-dbcnt** contains the number of lines read.

## Exercise 2: Creating an Executable Program

### Exercise Objectives

After completing this exercise, you will be able to:

- Create an executable program

### Business Example

You are responsible for the analyses in a development project. You want to create a list of flights.

#### Template:

none

#### Solution:

BC402\_INS\_FLIGHT\_LIST

#### Task:

Create a program that outputs all the flights for a specific airline. You want the user to enter an airline code on the input screen (selection screen) and - optionally - a constraint for the flight numbers. You want the corresponding flights to appear in an output template (list).

1. Go to the *Object Navigator* and create a program with type *Executable Program* with the name **ZBC402\_##\_FLIGHT\_LIST**.
2. Assign message class BC402 to the program as the standard message class.
3. Define a selection screen:  
Define an elementary input field for the airline code.  
Enable the user to enter one or more flight numbers.
4. Read the data from database table SFLIGHT. Use the user input on the selection screen to delimit the data. Define a suitable data object to save the data temporarily. Output a suitable error message if no data can be found for the selection criteria in the database.
5. Output the following data in a simple list:  
CARRID, CONNID, FLDAT, PRICE, CURRENCY, PLANETYPE,  
SEATSMAX, SEATSOCC

## Solution 2: Creating an Executable Program

### Task:

Create a program that outputs all the flights for a specific airline. You want the user to enter an airline code on the input screen (selection screen) and - optionally - a constraint for the flight numbers. You want the corresponding flights to appear in an output template (list).

1. Go to the *Object Navigator* and create a program with type *Executable Program* with the name **ZBC402\_##\_FLIGHT\_LIST**.
  - a) Carry out this step in the usual manner.
2. Assign message class BC402 to the program as the standard message class.
  - a) See the source code excerpt from the model solution.
3. Define a selection screen:  
Define an elementary input field for the airline code.  
Enable the user to enter one or more flight numbers.
  - a) See the source code excerpt from the model solution.
4. Read the data from database table SFLIGHT. Use the user input on the selection screen to delimit the data. Define a suitable data object to save the data temporarily. Output a suitable error message if no data can be found for the selection criteria in the database.
  - a) See the source code excerpt from the model solution.
5. Output the following data in a simple list:

*Continued on next page*

CARRID, CONNID, FLDATE, PRICE, CURRENCY, PLANETYPE,  
SEATSMAX, SEATSOCC

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

## Result

Source code excerpt from the model solution:

```
REPORT  bc402_ins_flight_list MESSAGE-ID bc402.

DATA: gt_flight TYPE TABLE OF sflight,
      gs_flight TYPE sflight.

PARAMETERS pa_car TYPE sflight-carrid.
SELECT-OPTIONS so_con FOR gs_flight-connid.

START-OF-SELECTION.

SELECT * FROM sflight INTO TABLE gt_flight
  WHERE carrid = pa_car
    AND connid IN so_con.

IF sy-subrc <> 0.
  MESSAGE e038.
ENDIF.

LOOP AT gt_flight INTO gs_flight.

  WRITE: /
    gs_flight-carrid,
    gs_flight-connid,
    gs_flight-fldate,
    gs_flight-price    CURRENCY gs_flight-currency,
    gs_flight-currency,
    gs_flight-planetype,
    gs_flight-seatsmax,
    gs_flight-seatsocc.

ENDLOOP.
```



## Lesson Summary

You should now be able to:

- Explain what Open SQL is
- Understand and program simple database read accesses with Open SQL
- Explain the difference between single record access, sequential access, and array fetch



## **Unit Summary**

You should now be able to:

- Explain the basic structures of ABAP programs
- Name the different types of ABAP programs
- Describe the components of the various ABAP program types
- Use the different modularization techniques available in ABAP
- Describe the type system in ABAP
- Name the respective advantages of global and local types
- Differentiate between flat and deep data objects
- Explain the visibility and validity of data objects
- Explain what Open SQL is
- Understand and program simple database read accesses with Open SQL
- Explain the difference between single record access, sequential access, and array fetch



Internal Use SAP Partner Only

# *Unit 2*

## **Program Calls and Memory Management**

### **Unit Overview**

In this unit, you learn about the techniques that let you call complete programs from within other programs.

In the second lesson, we turn to the runtime environment and learn how data objects are managed in working memory. In particular, we see how programs and program components are loaded subsequently during program calls that use modularization techniques, and how they are kept separate in memory.

Finally, we learn how access to ABAP memory and SAP memory, as well as shared objects, enable you to exchange data across program and user boundaries.

### **Unit Objectives**



After completing this unit, you will be able to:

- Explain how programs are called from within other programs
- Describe the various options for passing data on to the called program during program calls.
- Describe how the ABAP runtime environment executes programs
- Describe the importance of the runtime object
- Explain the terms "roll area" and "PXA"
- Explain memory management in the ABAP runtime environment
- Describe how dynamic data objects are managed
- Explain how "Boxed Components" are managed and when they should be used
- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program

### **Unit Contents**

Lesson: Calling Programs .....	37
--------------------------------	----

Exercise 3: Implementing Program Calls .....	45
Lesson: ABAP Runtime and Memory Management.....	54
Exercise 4: ABAP Memory and SAP Memory .....	85
Lesson: Shared Objects .....	94
Exercise 5: Optional: Using Shared Objects .....	113

# Lesson: Calling Programs

## Lesson Overview

In this lesson, you learn about the various techniques for calling complete programs synchronously. You learn how to insert and restart programs. Lastly, you learn how to pass data on to the called program during calls.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain how programs are called from within other programs
- Describe the various options for passing data on to the called program during program calls.

## Business Example

You want to call existing programs from within a new application. To do so, you familiarize yourself with the techniques for calling programs.

## Synchronous Program Calls

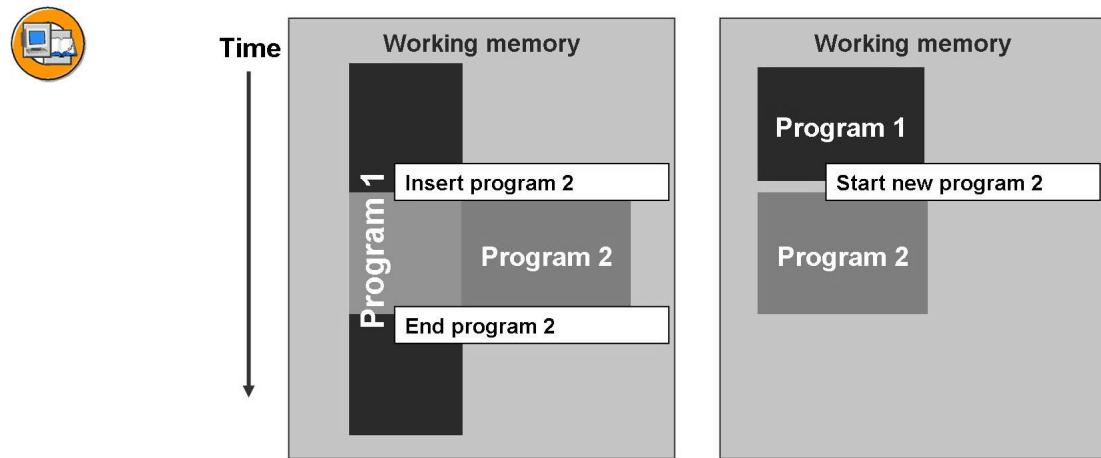
ABAP features two options for executing a program from within another program: the SUBMIT and CALL TRANSACTION (or LEAVE TO TRANSACTION) statements. The two statements differ in their effects: In the latter case, the calling program ends as soon as the called program is started. In the former case, the calling program waits until the called program is finished, and execution of the calling program continues with the statement after the program call. We therefore differentiate between the following terms:

### Insert

The process flow of the calling program pauses while the called program is being executed. After the called program is complete, the calling program is continued.

### New start

The calling program is terminated and the called program is started.

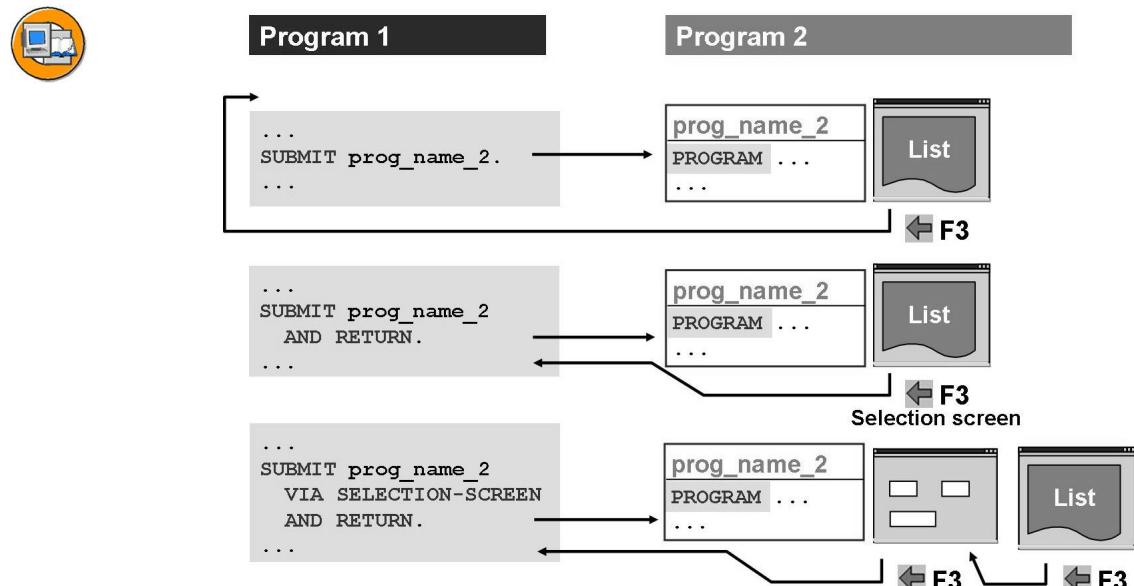
**Figure 21: Synchronous Program Calls**

Complete programs within a single user session can run only sequentially. This technique is also referred to as sequential calls. If you want to run activities in **parallel**, you have to use function modules.

→ **Note:** More information about this technique is available in other training courses and in the ABAP keyword documentation for the CALL FUNCTION ... STARTING NEW TASK ... statement.

## Synchronous Calls of Executable Programs

### The SUBMIT Statement

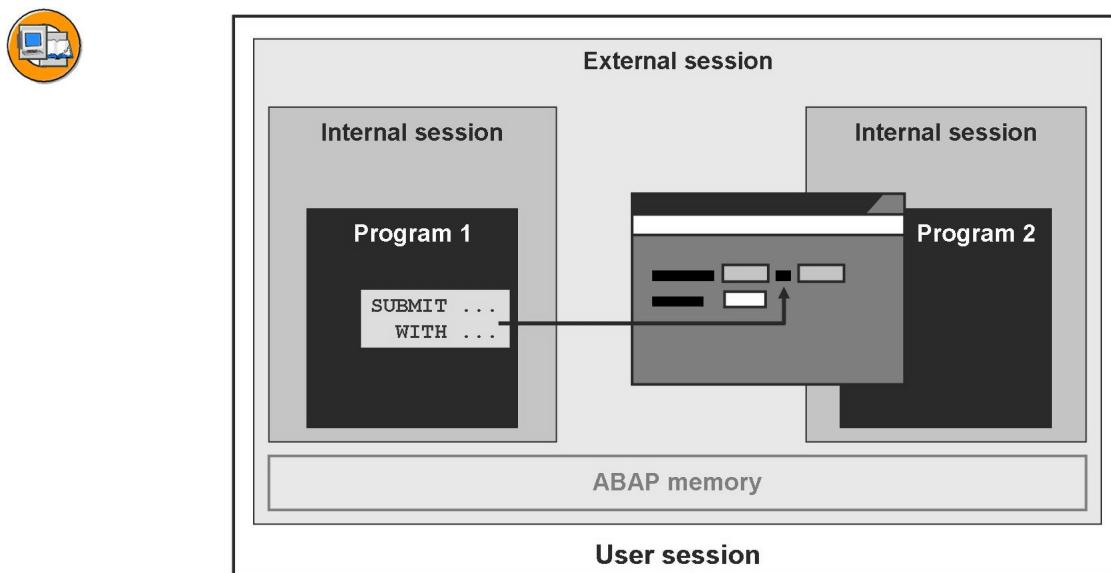
**Figure 22: Calling an Executable Program**

To start an executable (type 1) program, use the SUBMIT statement. If you use the VIA SELECTION-SCREEN addition, the standard selection screen of the called program (if any) is displayed.

If you use the AND RETURN addition, the system resumes processing with the first statement of the calling program after the SUBMIT statement, once the called program has finished.

→ **Note:** For more information, see the ABAP keyword documentation for the SUBMIT statement.

## Passing Data Using the Program Interface



**Figure 23: Passing Data Using the Program Interface**

When you call ABAP programs that have a standard selection screen, you can pass data for the input fields in the call. There are two ways to do this:

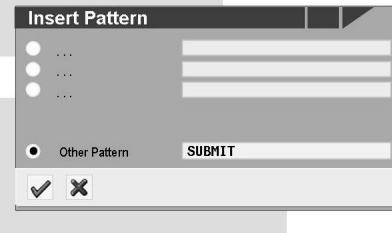
- By specifying a variant for the selection screen when you call the program
- By specifying specific values for the input fields when you call the program



```
DATA set {TYPE|LIKE} RANGE OF {type|dataobject}.
```

```
SUBMIT prog_name AND RETURN [VIA SELECTION-SCREEN]
WITH parameter {EQ|NE|...} val
WITH sel_opt {EQ|NE|...} val SIGN {'I'||'E'}
WITH sel_opt BETWEEN val1 AND val2 SIGN {'I'||'E'}
WITH sel_opt NOT BETWEEN val1 AND val2 SIGN {'I'||'E'}
WITH sel_opt IN set
...
```

```
MODULE user_command_0200 INPUT.
...
CASE save_ok.
  WHEN 'CTRYFR'.
    SUBMIT sapbc402_tabd hashed
      WITH pa_ctry = sdyn_conn-countryfr
      AND RETURN.
...
ENDCASE.
ENDMODULE.
```



**Figure 24: Preassignment of Input Fields**

The WITH addition in the SUBMIT statement allows you to assign values to the fields on a standard selection screen. The relational operators EQ, NE, ... and the inclusion operators I and E have the same meanings as with select options.

If you want to set several selections for a selection option, you can also use the RANGE type instead of individual WITH additions. This creates a selection table that you can fill as though it were a selection option. You then pass the whole table to the executable program.

If you want to display the standard selection screen when you call the program, use the VIA SELECTION-SCREEN addition.

Use the statement pattern in the *ABAP Editor* to insert a program call via SUBMIT. The pattern automatically supplies the names of the parameters and selection options that are available on the standard selection screen.



**Hint:** For more information about working with variants and about other syntax variants of the WITH addition, see the ABAP keyword documentation for the SUBMIT statement.

## Synchronous Transaction Calls

### The CALL TRANSACTION and LEAVE TO TRANSACTION Statements

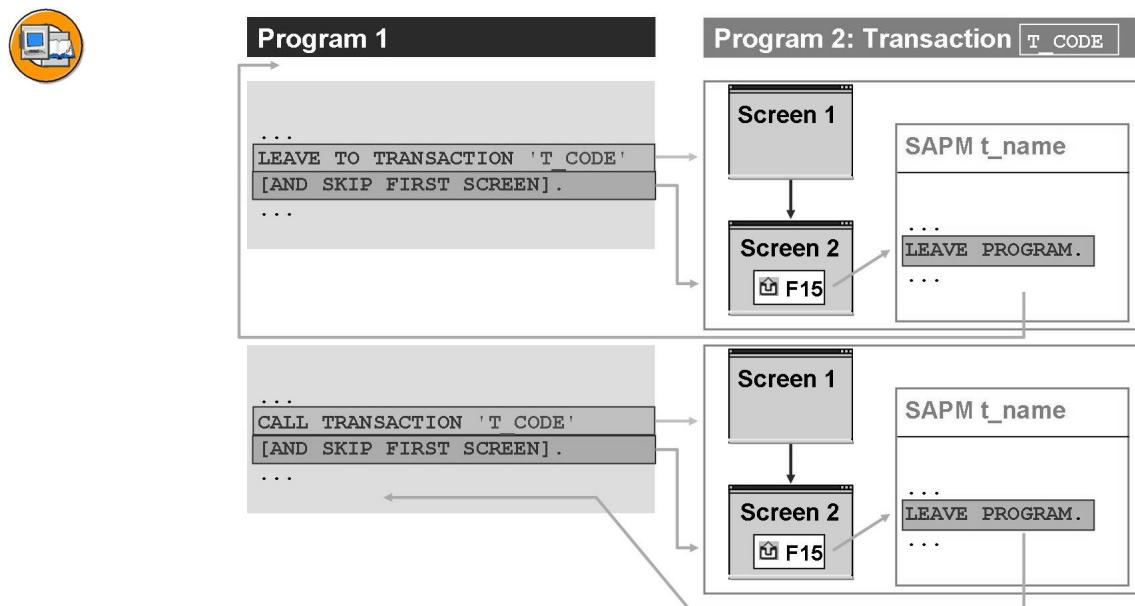


Figure 25: Calling a Transaction

If you use the **LEAVE TO TRANSACTION 'T\_CODE'** statement, the system terminates the current program and starts the transaction with transaction code **T\_CODE**. The statement is the equivalent of entering **/NT\_CODE** in the command field.

The **CALL TRANSACTION 'T\_CODE'** statement lets you insert a program that has a transaction code.

You can use the **LEAVE PROGRAM** statement to force the termination of a program. For cases where this statement is executed in a program that is called by **CALL TRANSACTION 'T\_CODE'** or **SUBMIT prog\_name AND RETURN**, you continue with the next statement after this call. In all other cases, the user returns to the application menu from which he or she started the program.

If you use the **... AND SKIP FIRST SCREEN** addition, the system does not display the contents of the first screen in the transaction.

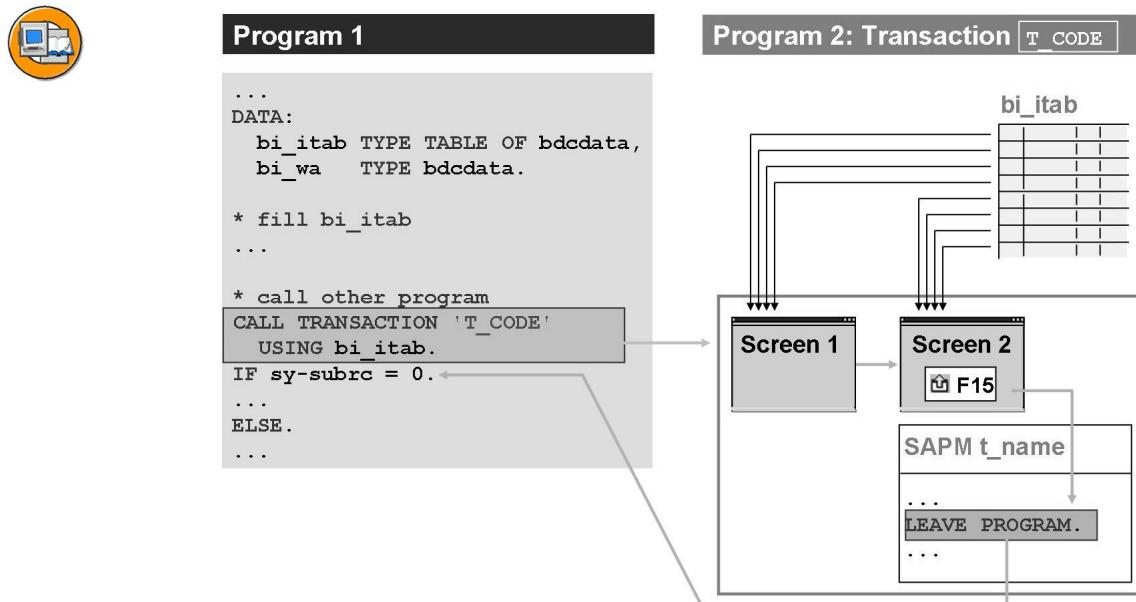


**Caution:** However, it does process the flow logic.

If you started a transaction using **CALL TRANSACTION 'T\_CODE'** and that transaction uses update techniques, you can use the **UPDATE . . .** addition to specify the update technique – asynchronous (default), synchronous, or local – that the program should use.

 **Note:** For more information, see training course BC414 (Programming Database Updates) and the SAP Library.

## Passing Data When Starting Transactions



**Figure 26: Passing Data Using an Internal Table**

When you call a transaction using the **CALL TRANSACTION 'T\_CODE' USING bi\_itab** statement, you can run transaction **T\_CODE** using the values from internal table **bi\_itab** in the screen fields. This internal table has to have the structure **BDCDATA** and be filled accordingly.

The **MODE** addition allows you to specify whether the screen contents should all be displayed (A – default setting), only when an error occurs (E), or not at all (N).

You can use the **MESSAGES INTO** addition to specify an internal table into which any system messages should be written. The corresponding internal table has to have the type of global structure **BDCMSGCOLL**.

You can find out if the transaction was executed successfully from the return code(SY-SUBRC).

This technique can be useful if:

- You are processing in dialog, but the input fields have not been filled using GET parameters
- You want to process the transaction “in the background”. In this case, you normally have to pass the function codes in the table as well.



Field name	Length	Description	Note when filling
program	40	Program name	Only in 1st record for screen
dynpro	4	Screen number	Only in 1st record for screen
dynbegin	1	First record	'X' for 1st record for screen, otherwise ' '
fnam	132	Field name	
fval	132	Field value	Case-sensitive!

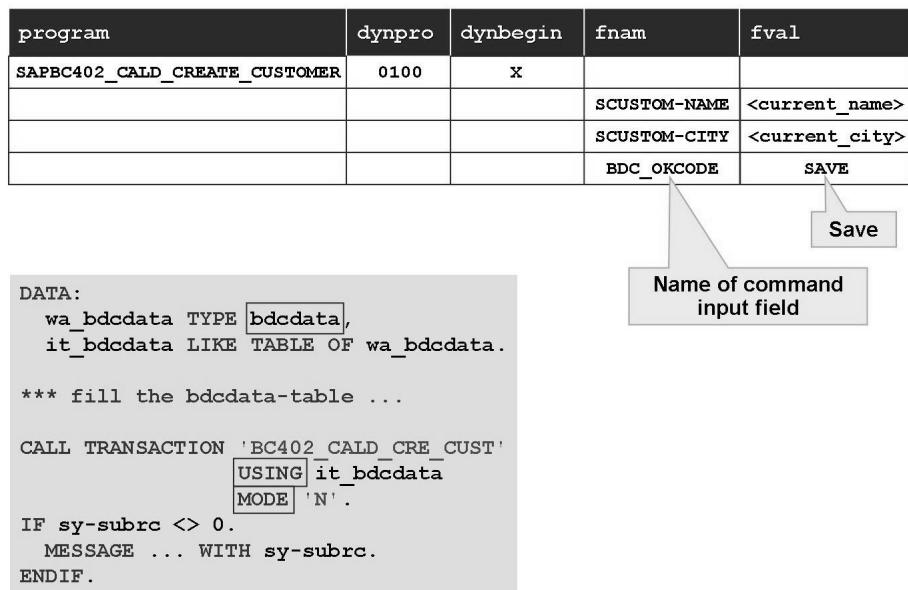
**Figure 27: Fields in the Global Type BDCDATA**

Filling the internal table in batch input format:

- Each screen that you want to fill and process automatically in the transaction must be identified by a line in which only the PROGRAM, DYNPRO, and DYNBEGIN fields are filled.
- After the record that identifies the screen, use a new BDCDATA record for each field you want to fill. These records use the FNAM and FVAL fields. You can fill the following fields:
  - Input/output fields (with data)
  - The command field BDC\_OKCODE (with a function code)
  - The field for cursor positioning BDC\_CURSOR (with a field name)

This technique is also one of the possible ways of transferring data from non-SAP systems. When you do this, the internal table with the structure BDCDATA must be filled completely.

For more information, see training course BC420 (Data Transfer) and the SAP Library.



**Figure 28: Data Transfer Using an Internal Table: Application Example**

The example above is an example for the transaction BC402\_CALD\_CTA\_U. When the user creates a new customer entry, the program calls transaction BC402\_CALD\_CRE\_CUST. This transaction has not implemented import from ABAP memory and its input fields are not set as GET parameters. The customer data is, therefore, passed using an internal table and processed “invisibly”.

If the operation is successful, the new customer record can be entered in the waiting list. The filled internal table in BDCDATA format is illustrated above.

At runtime, CURRENT\_NAME stands for the customer name from the input field and CURRENT\_CITY stands for the city. You address the command field using BDC\_OKCODE. Here you enter the function code that is triggered by the user choosing a function key, pushbutton, or menu option during the dialog flow (or by entering a code directly in the command field).

## Exercise 3: Implementing Program Calls

### Exercise Objectives

After completing this exercise, you will be able to:

- Implement calls of executable programs
- Implement transaction calls
- Pass data on to executable programs and transactions

### Business Example

An existing program displays a list of data. You want to enhance the program: When a user double-clicks a line, it should take them to an existing report or dialog transaction.

#### Template:

BC402\_PCT\_CONN\_LIST

#### Solution:

BC400\_PCS\_CONN\_LIST

#### Task 1:

Copy program BC402\_PCT\_CONN\_LIST and all its components to the name **ZBC402##\_CONN\_LIST**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the template.
2. Activate and test the program.

*Continued on next page*

3. Analyze the source code. How is the display of the flight connections implemented?

---

---

---

4. The program is already prepared to handle double-click events. Which processing block do you have to enhance to do so?

---

---

---

### Task 2:

Implement the handler method that runs when a user double-clicks a flight connection. Make sure a list of flights is returned for the clicked connection if the user does not double-click in the CARRID column. To do so, use executable program **BC402\_INS\_FLIGHT\_LIST** (or your own program, **ZBC402\_##\_FLIGHT\_LIST**).

1. Implement the `on_double_click` method of the local class `lcl_handler`. The `COLUMN` import parameter contains the information as to which column the user clicked. Use the `SUBMIT` statement to call the executable program.
2. Make sure users can return to the display of flight connections after displaying the dates.
3. Supply the interface of the called program with the key values of the clicked flight connection. To do so, determine the names of the input fields on the selection screen.

### Task 3:

Implement the handler method that runs when a user double-clicks a flight connection. Make sure details for the airline are output for the clicked connection if the user double-clicks in the CARRID column. To do so, call dialog transaction **BC402MCAR**.

1. Call the transaction so it is inserted synchronously.
2. Pass the airline ID of the clicked connection on to the transaction. To do so, use the `USING` addition. Determine the program name, screen number, and field label of the input field on the initial screen of transaction BC402MCAR.

*Continued on next page*

3. In addition, pass the function code that is linked with the pushbutton on the initial screen on to the transaction.
4. Make sure the transaction is processed without screen display. To do so, use the MODE addition with a suitable value.

## Solution 3: Implementing Program Calls

### Task 1:

Copy program BC402\_PCT\_CONN\_LIST and all its components to the name **ZBC402\_##\_CONN\_LIST**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the template.
  - a) Carry out this step in the usual manner.
2. Activate and test the program.
  - a) Carry out this step in the usual manner.
3. Analyze the source code. How is the display of the flight connections implemented?

**Answer:** Using an instance of class cl\_salv\_table. The static factory method is called to generate an instance of the class and pass it the table with the data. The display method shows the data in an ALV grid control.

### Related Information

Class cl\_salv\_table is a shared, object-oriented wrapper for conventional ALV lists (flat and hierarchical) and ALV grid controls. It is available in *NetWeaver Application Server 6.40* and later.

4. The program is already prepared to handle double-click events. Which processing block do you have to enhance to do so?

**Answer:** The on\_double\_click method of the local class lcl\_handler.

### Task 2:

Implement the handler method that runs when a user double-clicks a flight connection. Make sure a list of flights is returned for the clicked connection if the user does not double-click in the CARRID column. To do so, use executable program **BC402\_INS\_FLIGHT\_LIST** (or your own program, **ZBC402\_##\_FLIGHT\_LIST**).

1. Implement the on\_double\_click method of the local class lcl\_handler. The COLUMN import parameter contains the information as to which column the user clicked. Use the SUBMIT statement to call the executable program.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

2. Make sure users can return to the display of flight connections after displaying the dates.
  - a) AND RETURN addition; see source code excerpt from the model solution.
3. Supply the interface of the called program with the key values of the clicked flight connection. To do so, determine the names of the input fields on the selection screen.
  - a) See the source code excerpt from the model solution.

### Task 3:

Implement the handler method that runs when a user double-clicks a flight connection. Make sure details for the airline are output for the clicked connection if the user double-clicks in the CARRID column. To do so, call dialog transaction **BC402MCAR**.

1. Call the transaction so it is inserted synchronously.
  - a) CALL TRANSACTION. See source code excerpt from the model solution.
2. Pass the airline ID of the clicked connection on to the transaction. To do so, use the USING addition. Determine the program name, screen number, and field label of the input field on the initial screen of transaction BC402MCAR.
  - a) See the source code excerpt from the model solution.
3. In addition, pass the function code that is linked with the pushbutton on the initial screen on to the transaction.
  - a) See the source code excerpt from the model solution.
4. Make sure the transaction is processed without screen display. To do so, use the MODE addition with a suitable value.
  - a)

### Result

Source code excerpt from the model solution:

```
REPORT  bc402_pcs_conn_list MESSAGE-ID bc402.

TYPES: BEGIN OF gty_s_conn,
        carrid    TYPE spfli-carrid,
        connid    TYPE spfli-connid,
        cityfrom  TYPE spfli-cityfrom,
        cityto    TYPE spfli-cityto,
```

*Continued on next page*

```

        deptime  TYPE spfli-deptime,
        arrtime  TYPE spfli-arrtime,
        period    TYPE spfli-period,
      END OF gty_s_conn.

TYPES gty_t_conn TYPE STANDARD TABLE OF gty_s_conn
      WITH NON-UNIQUE DEFAULT KEY..

DATA gt_conn TYPE gty_t_conn.

DATA gv_msg TYPE string.

DATA: go_alv TYPE REF TO cl_salv_table,
      go_evt TYPE REF TO cl_salv_events_table,
      go_fct TYPE REF TO cl_salv_functions_list,
      gx_msg TYPE REF TO cx_salv_msg.

*-----*
*      CLASS lcl_handler DEFINITION
*-----*
*
*-----*
CLASS lcl_handler DEFINITION.

PUBLIC SECTION.

CLASS-METHODS:
      on_double_click FOR EVENT double_click
          OF if_salv_events_actions_table
          IMPORTING row column.

ENDCLASS.           "lcl_handler DEFINITION

*-----*
*      CLASS lcl_handler IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_handler IMPLEMENTATION.

METHOD on_double_click.

DATA: lt_bdc TYPE TABLE OF bdcdata,

```

*Continued on next page*

```

ls_bdc TYPE          bdcdata.

DATA ls_conn LIKE LINE OF gt_conn.

READ TABLE gt_conn INTO ls_conn INDEX row.

CASE column.
  WHEN 'CARRID'.

    ls_bdc-program = 'SAPMBC402_IND_CARRIER'.
    ls_bdc-dynpro   = '0100'.
    ls_bdc-dynbegin = 'X'.
    APPEND ls_bdc TO lt_bdc.

    CLEAR ls_bdc.
    ls_bdc-fnam = 'BC402_S_CARRIER-CARRID'.
    ls_bdc-fval = ls_conn-carrid.
    APPEND ls_bdc TO lt_bdc.

    CLEAR ls_bdc.
    ls_bdc-fnam = 'BDC_OKCODE'.
    ls_bdc-fval = 'GO'.
    APPEND ls_bdc TO lt_bdc.

    CALL TRANSACTION 'BC402MCAR'
      USING lt_bdc
      MODE 'E'.

  WHEN OTHERS.

    SUBMIT bc402_ins_flight_list
      AND RETURN
      WITH pa_car EQ ls_conn-carrid
      WITH so_con EQ ls_conn-connid.

ENDCASE.

ENDMETHOD.           "on_double_click

ENDCLASS.           "lcl_handler IMPLEMENTATION

*-----*
START-OF-SELECTION.

```

*Continued on next page*

```
SELECT carrid connid cityfrom cityto
      deptime arrtime period
   FROM spfli
  INTO TABLE gt_conn.

TRY.
  CALL METHOD cl_salv_table=>factory
    IMPORTING
      r_salv_table = go_alv
    CHANGING
      t_table      = gt_conn.
  CATCH cx_salv_msg INTO gx_msg.
    gv_msg = gx_msg->get_text( ).
    MESSAGE gv_msg TYPE 'E'.
ENDTRY.

go_evt = go_alv->get_event( ).

SET HANDLER lcl_handler=>on_double_click
  FOR go_evt.

go_alv->display( ).
```



## Lesson Summary

You should now be able to:

- Explain how programs are called from within other programs
- Describe the various options for passing data on to the called program during program calls.

# Lesson: ABAP Runtime and Memory Management

## Lesson Overview

This lesson gives you a detailed look at how the ABAP runtime environment executes a program. We pay special attention to memory management and how the memory of the various program units is delimited when modularization units are called. In this context, you learn about the ABAP memory and SAP memory and how you can use these two techniques for transient data exchange across program boundaries.

Lastly, you learn how data objects with variable length (dynamic data objects) are managed in memory. This should help you avoid unnecessary consumption of memory and CPU time when working with such data objects.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe how the ABAP runtime environment executes programs
- Describe the importance of the runtime object
- Explain the terms "roll area" and "PXA"
- Explain memory management in the ABAP runtime environment
- Describe how dynamic data objects are managed
- Explain how "Boxed Components" are managed and when they should be used

## Business Example

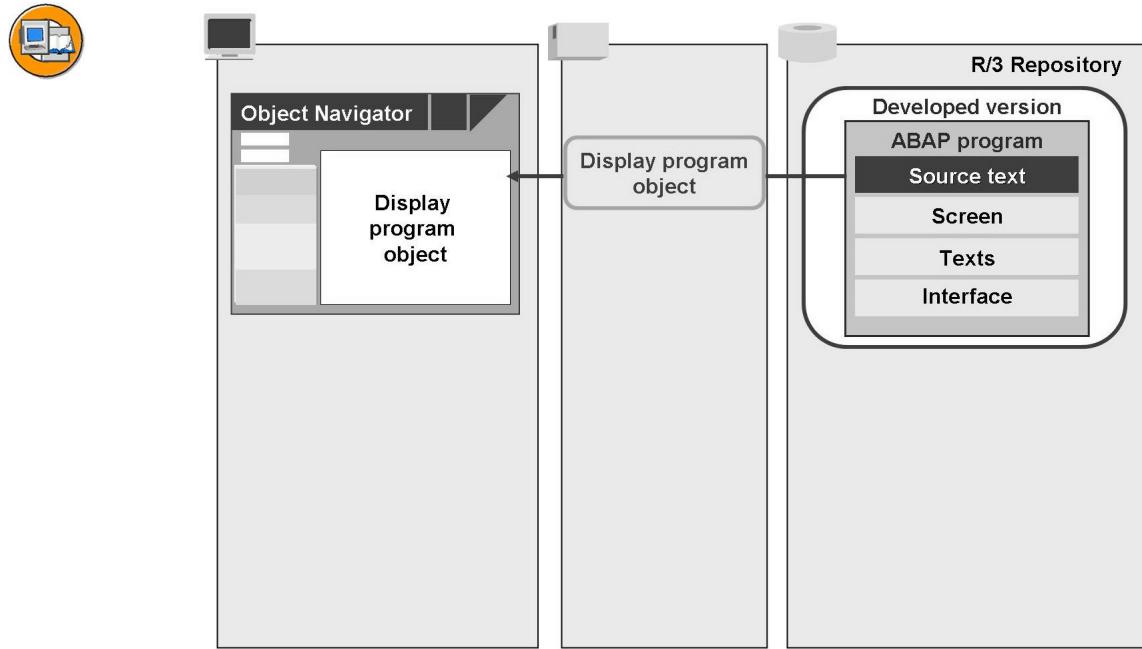
You want to develop complex applications that include program calls, modularization, and dynamic data objects. To ensure your applications have a robust, high-performance design, you want to find out how the runtime environment and memory management work in detail. You also want to learn about the options presented by ABAP memory and SAP memory, to exchange data across program boundaries during program calls.

## The ABAP Runtime System

The ABAP runtime system is the core of the *SAP NetWeaver Application Server ABAP*. It is a hardware-independent, operating system-independent, and database-independent platform (**virtual machine**) for running ABAP programs. The processes in the ABAP runtime environment control the execution of ABAP programs by calling the processing blocks of the program.

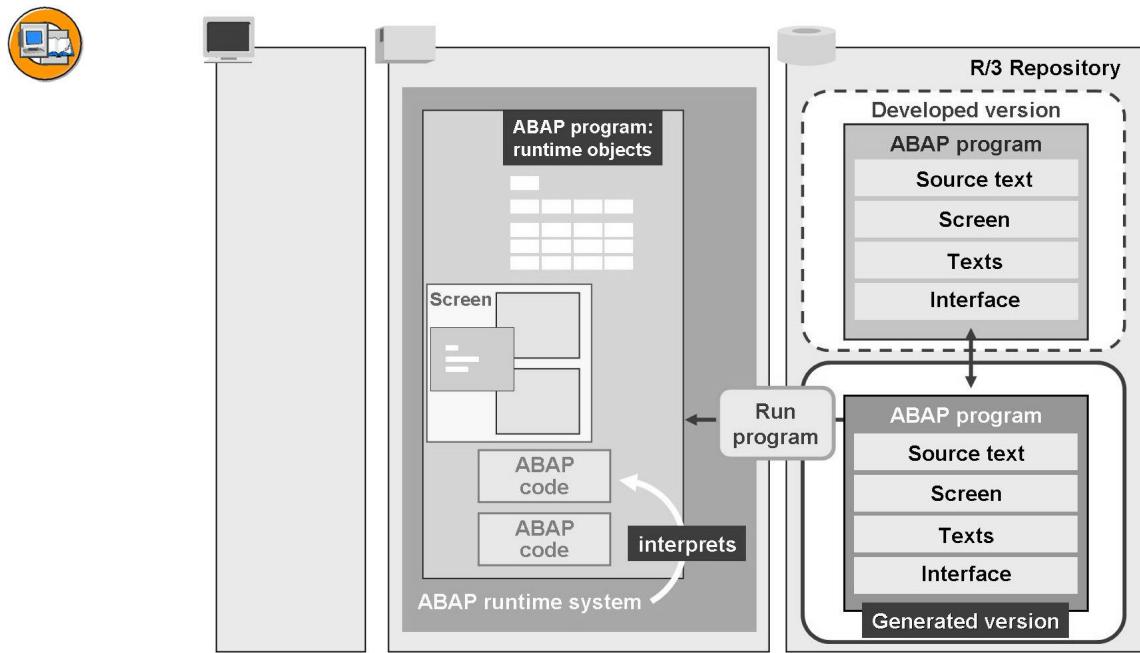
## ABAP Compiler and Runtime Objects

In principle, development objects are stored in the Repository in the database in the form in which they are developed. These are the only versions that can be displayed in the *Workbench*.



**Figure 29: Developed Version**

For a program to run in the ABAP runtime environment, it first has to be compiled (or, in the ABAP terminology: generated) by the *ABAP Compiler*.



**Figure 30: Generated Version**

When a development object is generated, a runtime object, or **LOAD**, is generated. The LOAD consists largely of byte code whose statements are linked with C functions. To avoid having to generate programs every time they run, program LOADs can be saved in the database and buffered on the application server.

The LOAD is generated **automatically** when the program is called if none exists yet or if the LOAD in the database or the program buffer is obsolete. You can also generate programs explicitly, for example, in the *ABAP Editor* with menu path *Program → Generate*.

A saved or buffered LOAD is considered obsolete when:



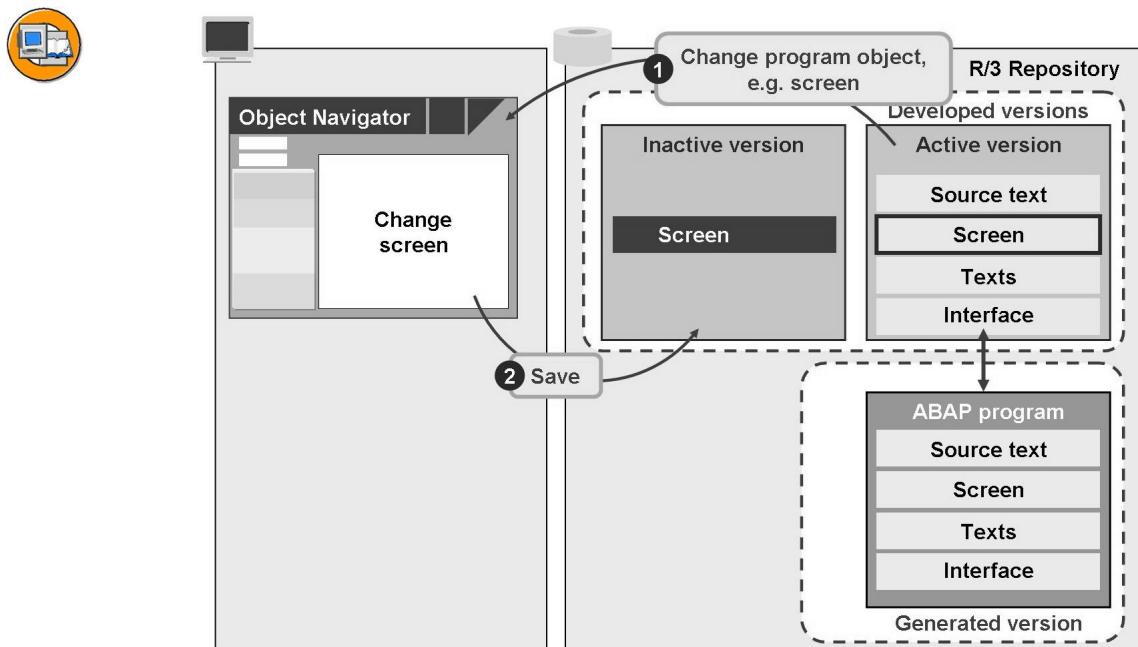
- The developed version of the program was changed, or
- An object that the program uses was changed in the *Data Dictionary*.



**Hint:** Transports between SAP systems always involve developed versions only. After an import into a target system, all affected runtime objects are discarded. New LOADs are not generated automatically; they are generated the first time a user calls the corresponding program (directly or indirectly). After a large import, users may temporarily experience wait times and lower performance until all the required LOADs are generated again. To avoid this, you can start the SAP load generator (transaction code SGEN) after the import.

## The Concept of Inactive Sources

If you change a development object and then save it, the system creates an active version and an **inactive version**.

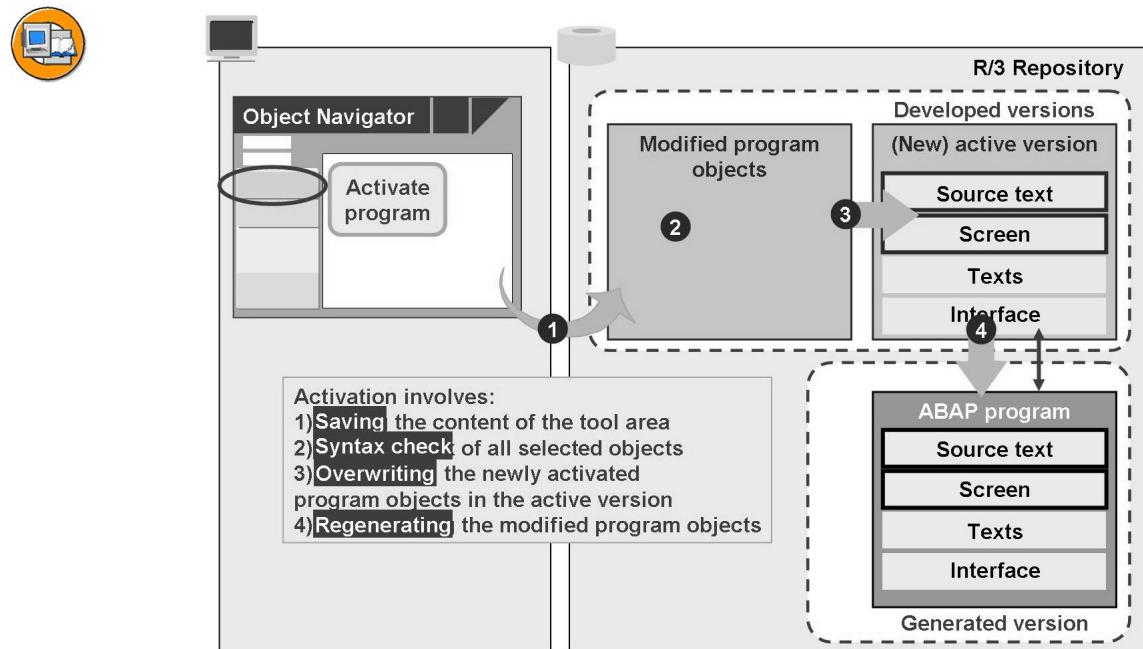


**Figure 31: Active and Inactive Versions**



**Hint:** Saving an inactive version has **no** effect on the program LOAD. Accordingly, the syntax of the inactive version does not necessarily have to be correct. You can also save an interim, nonexecutable version of the development inactively. The active version remains available as an executable version, however.

You can toggle the view between active and inactive versions as necessary, however, you can only edit the inactive version. The syntax check is carried out on the version in development, not the generated version. This means it can be performed on the inactive version. When running the source code for executable programs that are started from the Editor, the system goes back to the inactive version. In this case, a **temporary generated version** is created, but is not buffered or stored in the database. In all other cases, the system always accesses the active version when the program is generated.



**Figure 32: Activating a Program**

All inactive objects that are assigned to your user ID are listed in your worklist. The following steps occur when you activate an object (or several objects):

1. The object is saved – that is, a new inactive version is created.
2. This is subject to a syntax check.
3. The system then overwrites the current active version. The previously inactive version is now the active version. There is no longer an inactive version.
4. The system creates a new runtime version. The LOAD is updated.

### PXA and Roll Area

When the runtime system generates a new runtime object (the LOAD) of a program or loads it from the database, the program is placed in the working memory of the application server. In the process, the modifiable parts of the program are treated differently than the nonmodifiable parts.

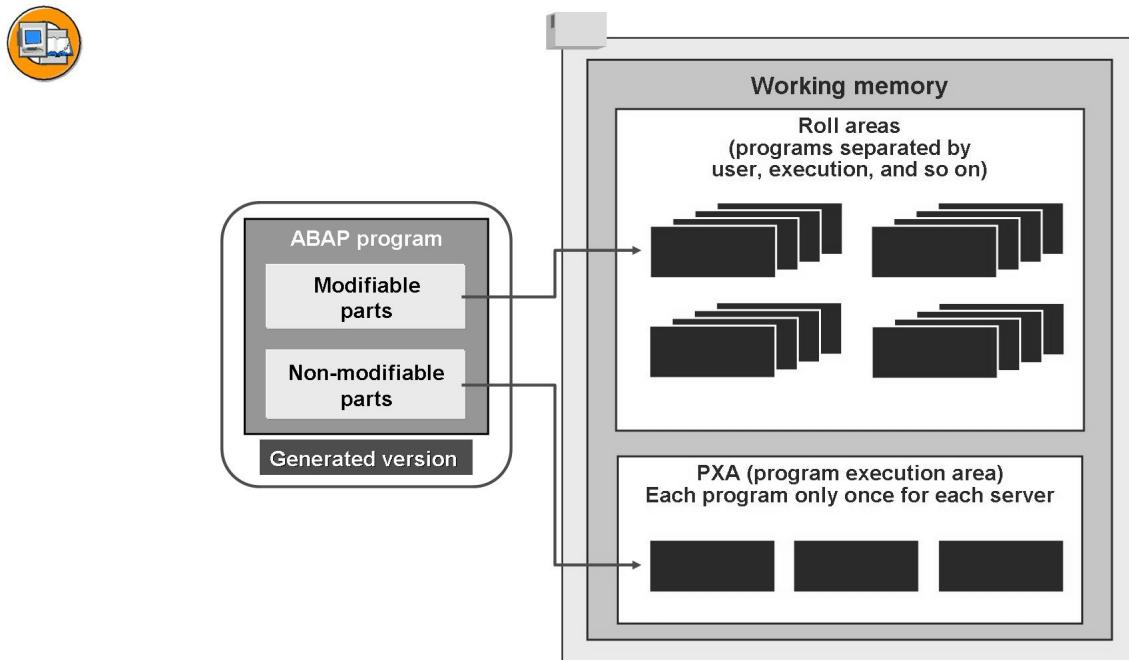


#### The nonmodifiable parts of a program are:

- Byte code for statements
- Values of constants and literals
- Program texts
- Screen definitions

The modifiable parts of a program are:

- Data objects (variables)



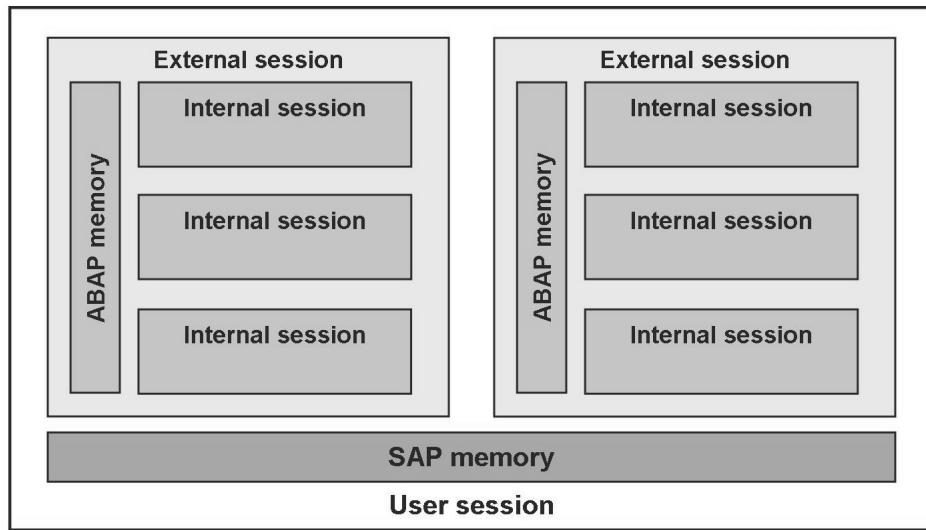
**Figure 33: Loading a Runtime Object to Working Memory**

The **nonmodifiable parts of the program** are stored in the program execution area (PXA) and buffered there as long as possible. This memory area is shared by all users on the application server and only exists once on each server. After a program is executed, its byte code remains buffered in the PXA as long as possible, to avoid having to load it from the database again during the next execution.

The **modifiable parts of the program** are created from scratch in the memory during every execution (in a memory space called the roll area). If a program is executed several times concurrently (by the same user or different users), each execution has its own roll area. This ensures that the programs can be executed independently of one another.

## Memory Management from the Program Perspective

The way in which main memory is organized from the program's perspective can be represented in the following simple model:



**Figure 34: Logical Memory Model**

There is a distinction between internal and external sessions:

#### External session (main session)

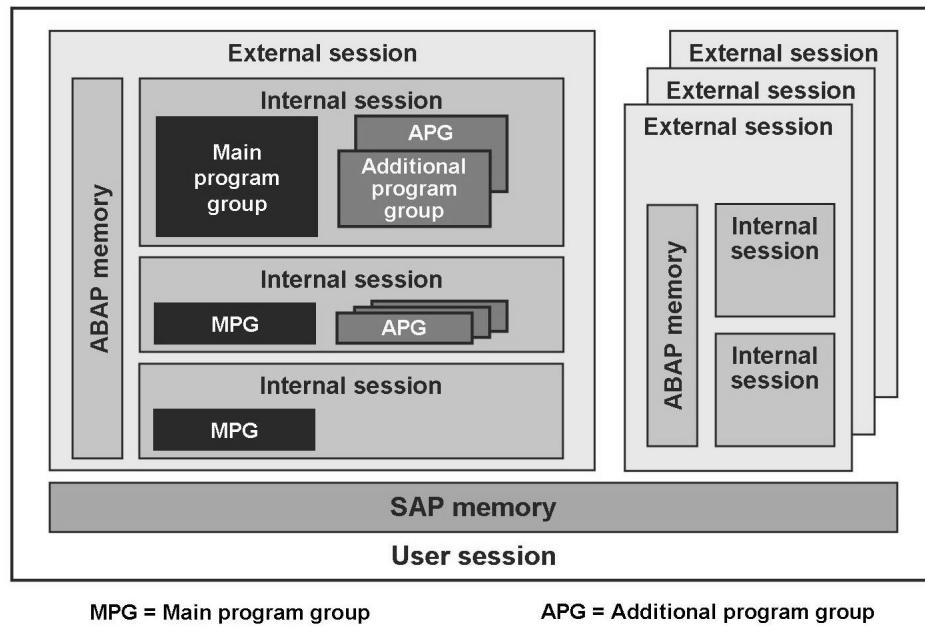
Generally, an external session corresponds to an SAPGUI window. You can create a new session with System → New Session or by calling `/o<t_code>`. Up to 16 external sessions are possible in Release 7.0 and later (the maximum was 6 in earlier releases). The maximum number available in a given system is defined by system parameter rdisp/max\_alt\_modes, whose default value is 6.

#### Internal session

External sessions are subdivided into internal sessions (placed in a stack). Each call of an ABAP program (SUBMIT, CALL TRANSACTION, and so on) generates a new internal session. A maximum of **nine internal sessions** can exist in each external session; they must all belong to the same call stack.



**Hint:** One roll area is reserved for each internal session.



MPG = Main program group

APG = Additional program group

**Figure 35: Program Groups**

When a program calls external modularization units, the corresponding programs are loaded into the same internal session. The programs within an internal session form **program groups**. When an internal session is opened with a program call, the **main program group** is always created. There can also be one or more **additional program groups**. Each program group contains one main program and possibly other programs as well.

A program's data objects are only visible within its program group, which means programs in different program groups do not share any data objects.

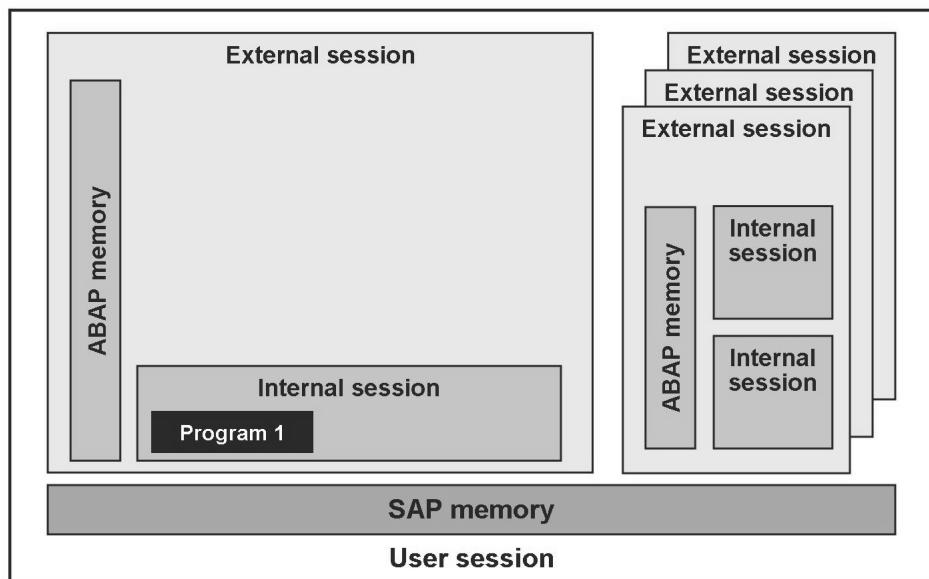
All programs (and objects) of an internal session can use instances of classes. An object continues to exist as long as there are users - that is, references - for it.

## Memory Management for Program Calls

The following illustrates how a stack of internal sessions inside an external session changes with various program calls.

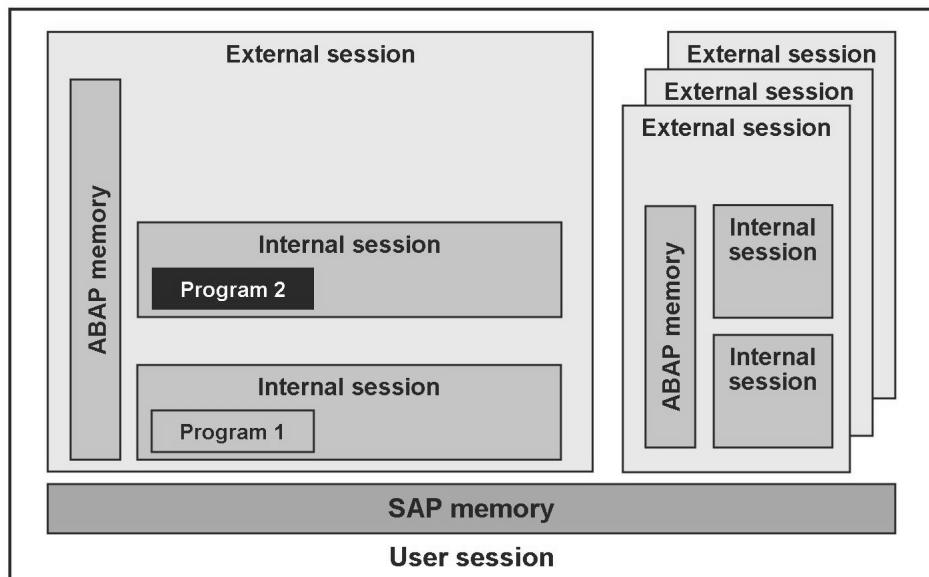
### Inserting a Program

The starting point for the first case is the execution of the **SUBMIT . . . AND RETURN** or **CALL TRANSACTION** statements within a program:



**Figure 36: Before Inserting a Program**

When you insert a program, the system creates a new internal session, which contains the new program context. The new session is placed in the stack. The program context of the calling program remains in the stack.



**Figure 37: The Inserted Program Runs**

When the called program finishes, its internal session (the top one in the stack) is deleted. Processing is resumed in the next-highest internal session in the stack.

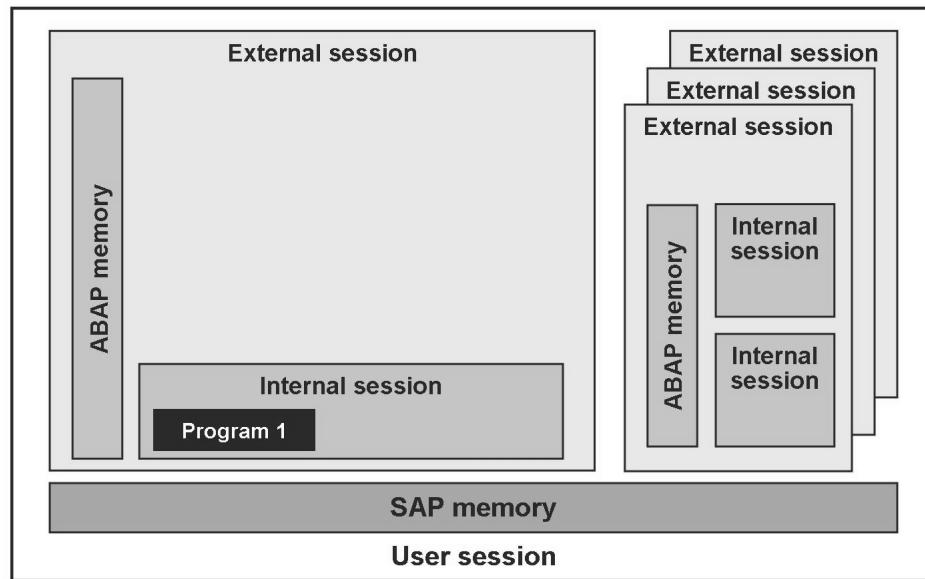


Figure 38: After the Inserted Program Ends

### Starting a New Executable Program

When you end a program and start a new one, there is a distinction between calling an executable program and calling a transaction, with regard to memory areas.

The starting point for the second case is the execution of a **SUBMIT** statement within a program:

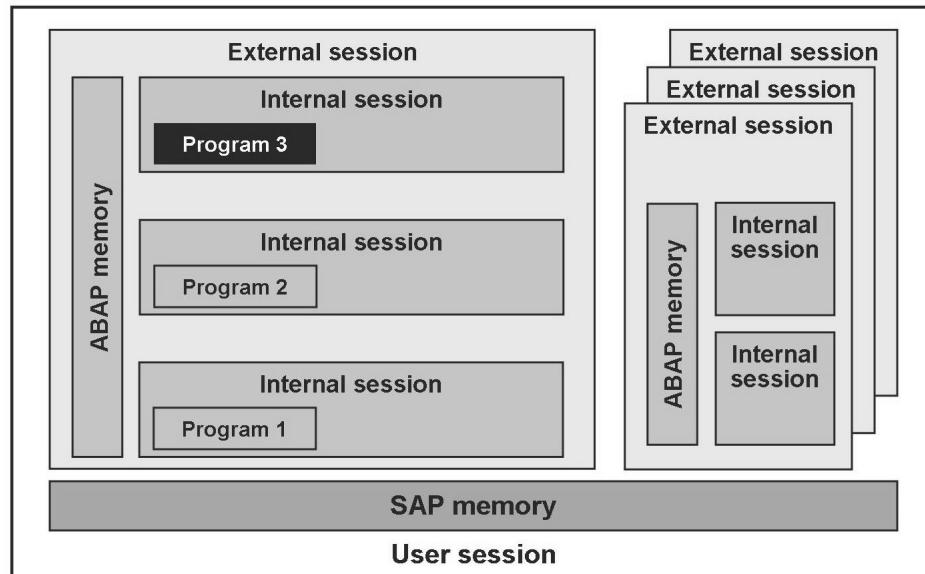


Figure 39: Before Starting a New Executable Program



If you call an executable program using its program name (and end the calling program), the system destroys the internal session of the program that you are terminating (the top one from the stack). The system creates a new internal session, which contains the program context of the called program. The new session is placed in the stack. Any program contexts that have been created before are retained. Therefore, the topmost internal session in the stack is ultimately replaced.

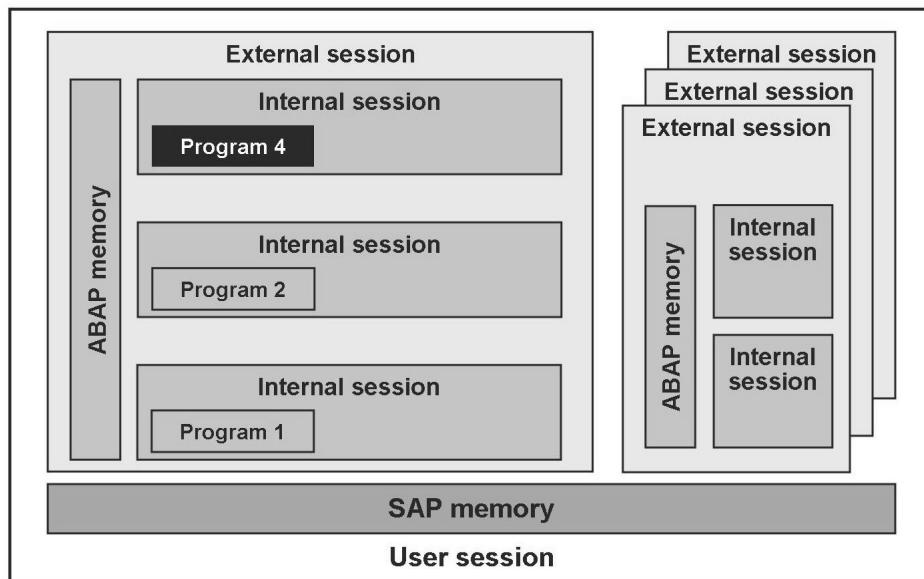
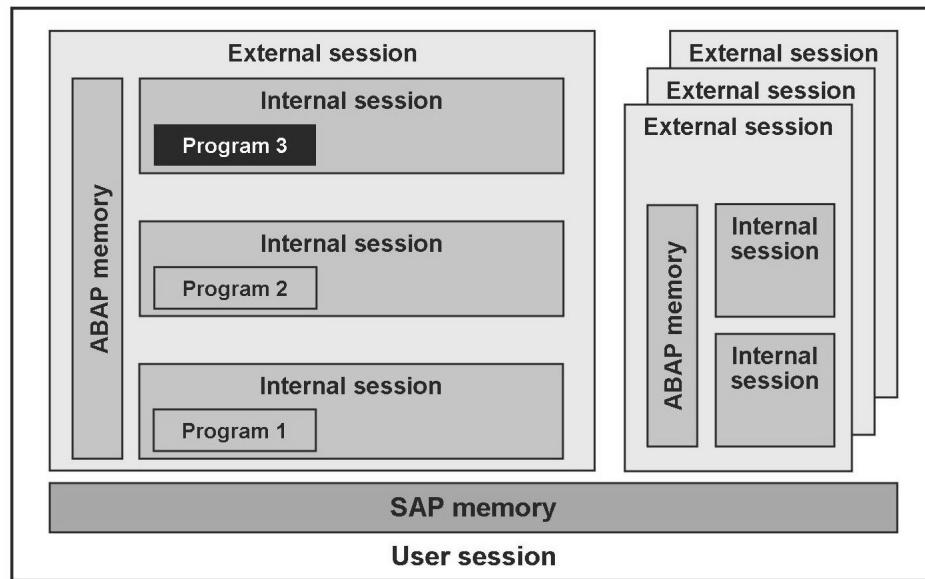


Figure 40: The New Executable Program Runs

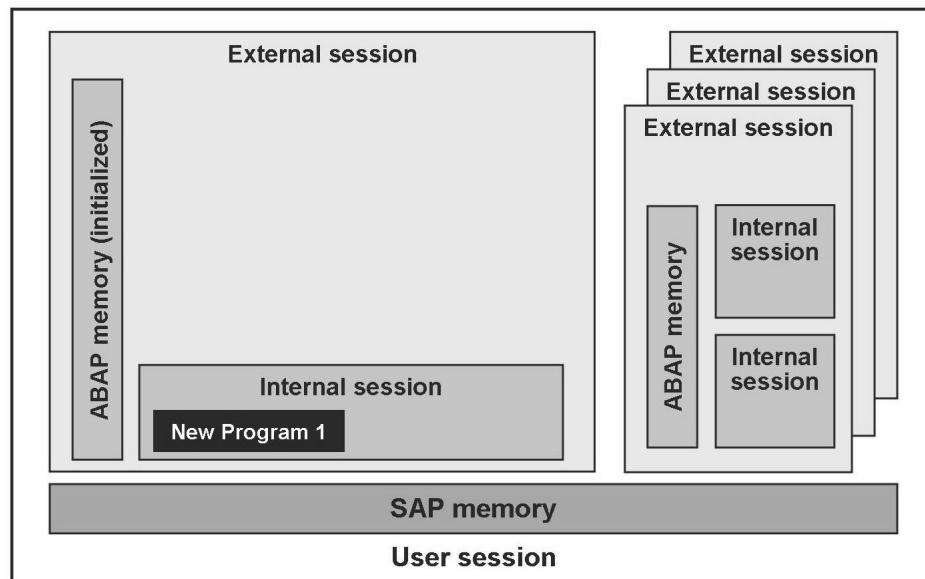
### Starting a New Transaction

The starting point for the third case is the execution of a **LEAVE TO TRANSACTION** statement within a program:



**Figure 41: Before Starting a New Transaction**

If you start a program using its transaction code, all of the internal sessions in the stack are destroyed. The system creates a new internal session, which contains the program context of the called program.



**Figure 42: The New Transaction Runs**

This means the complete stack of internal sessions is initialized, which also means the ABAP memory is initialized after the call.

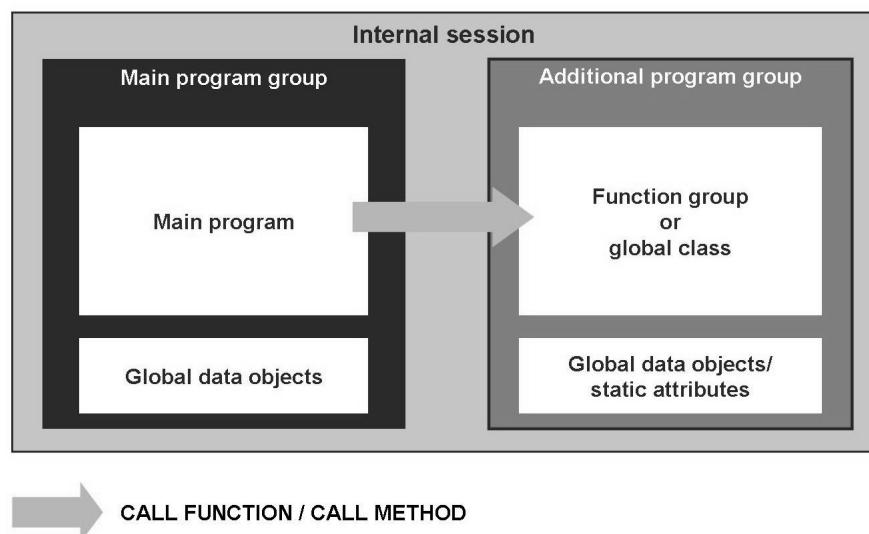
## Memory Management for Calls of Function Modules and Methods

Cross-program modularization - that is, calling function modules and using global classes - is not implemented using internal sessions. Instead, the program groups play a central role.



### Impact of external procedure calls for program type

- Function group (type F)
- Class pool (type K)



**Figure 43: Main/Additional Program Groups**

When an internal session is opened, the **main program group** is always created. Any number of additional program groups can also be created in the same internal session. Each program group has exactly one **main program**.

When a function module from a function group that has not been loaded yet or a method from a global class that has not been used yet is called, an additional program group is created. Each function group and each global class is loaded to a given internal session **exactly once**. The significance of this is illustrated in the following examples:

## Function Groups and Classes in Different Internal Sessions

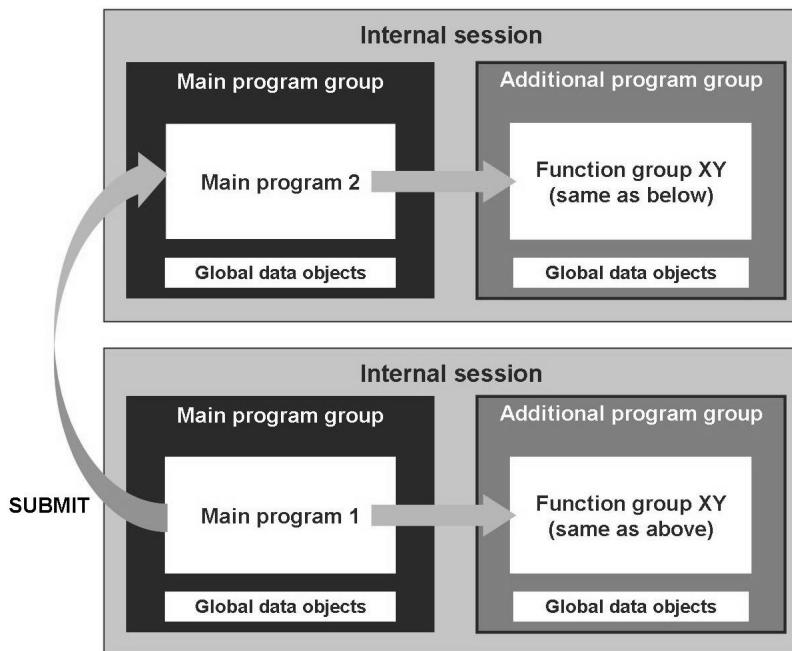
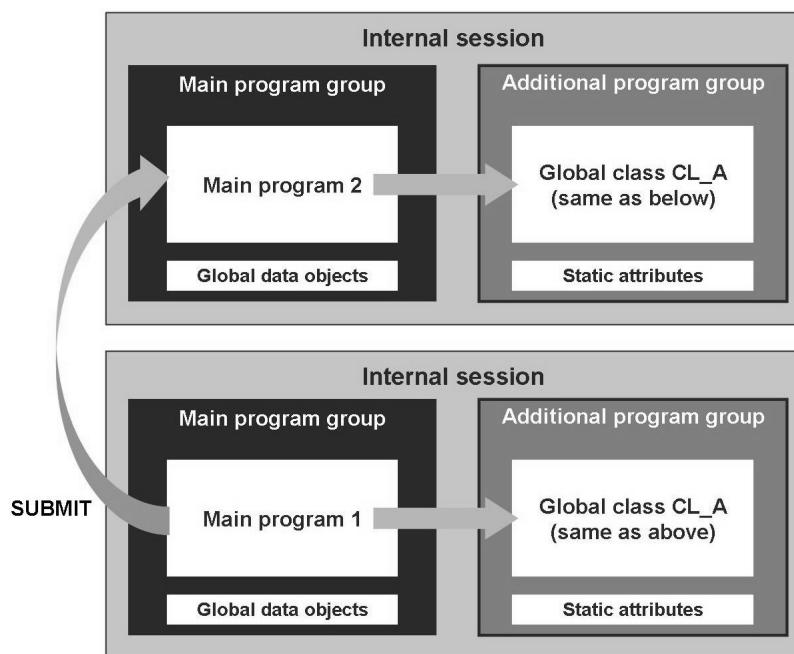


Figure 44: Same Function Group in Different Internal Sessions

If two programs that are running in **different** internal sessions call function modules from the same function group, the function group is loaded separately into each of the two internal sessions. The global data of the function group is created separately in each internal session. Therefore, there is no possibility or risk of the two main programs exchanging data with one another through the function group.



**Figure 45: Same Class in Different Internal Sessions**

Likewise, when a global class is used by programs that run in **different** internal sessions, the global class is also created separately in each internal session. The static attributes of the class can have different values in the different internal sessions.

## Function Groups and Classes in the Same Internal Session

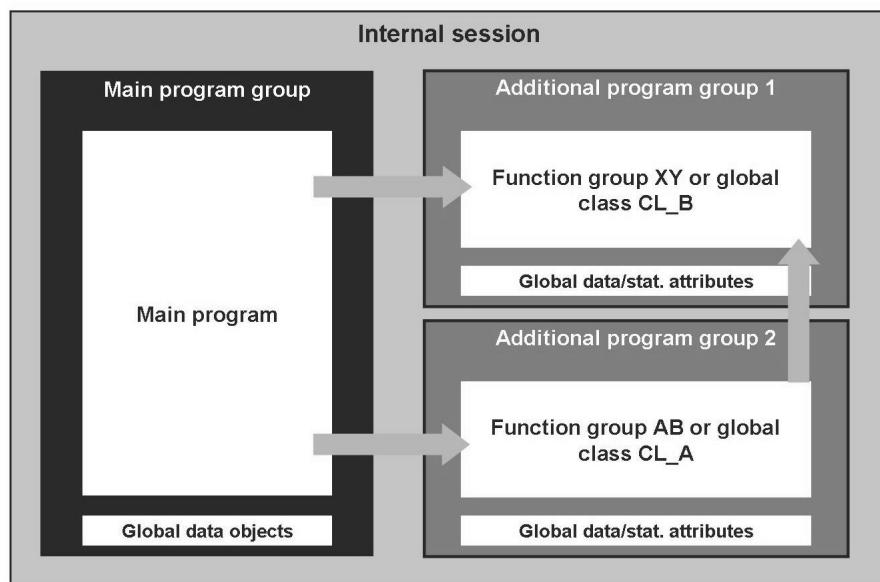


Figure 46: Same Function Group/Class within an Internal Session

If two programs run in the **same** internal session (for example, an executable program and the AB function group it addresses) and both programs address the same XY function group, then function group XY is only loaded into the internal session once. Therefore, the main program and function group AB can exchange data via function group XY. The same applies for global classes.



**Caution:** In practice, this situation can result in errors now and again, because the main program and function group AB (or class CL\_A) affect each other inadvertently. The main program might buffer data in function group XY, for example, but function group AB or class CL\_A could unintentionally overwrite the data or initialize it. This can result in program terminations and even data inconsistencies.

The causes of such errors are difficult to find - particularly if function group AB or class CL\_A are not used by the main program directly, but instead through other function groups or classes.



**Hint:** If the shared program is a global class (class CL\_B in the diagram), there is only a risk of unintended interaction with regard to the static attributes. The two programs used only see the same instances (and therefore the instance attribute values) of a class if they specifically exchange references to these instances.

An elegant way of avoiding unintended interaction for global classes is to use instance attributes instead of static attributes and have every user of the class generate a separate instance.

## External Subroutine Calls



### Effects of external subroutine calls with program types

- Executable program (type 1)
- Module pool (type M)
- Subroutine pool (type S)

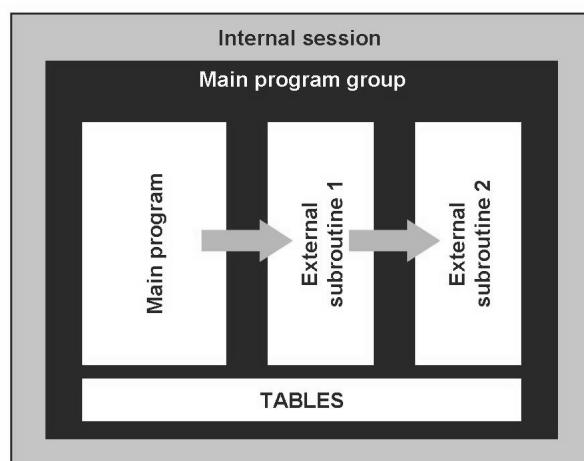


Figure 47: Several Programs in the Same Program Group

When a subroutine is called externally, **no new program group** is generated in the internal session. Instead, the main program of this subroutine is also loaded into the calling program's existing program group.



**Hint:** Subroutines of a function group are an exception to this: When this type of subroutine is called externally, the corresponding function group is loaded into an additional program group.

If two programs are in the same program group, the developer must observe the following aspects:

All the programs in a program group share the interface work areas that are declared with TABLES, TABLES, NODES, and COMMON PART.

Within a program group, CALL SCREEN can only be used to call screens of the **main program**.



**Caution:** When you use external subroutine calls, you always run the risk that data objects from one program can be shared with other programs. While this may be intended in some exceptional cases, you should avoid it wherever possible under the aspects of encapsulation and ease of maintenance. Therefore, avoid using external subroutine calls altogether.

## Data Transfer

When programs run in different program groups, internal sessions, or external sessions, they do not share any data objects. Nonetheless, various options are available for exchanging data between these programs.

### Overview of Data Transfer Options

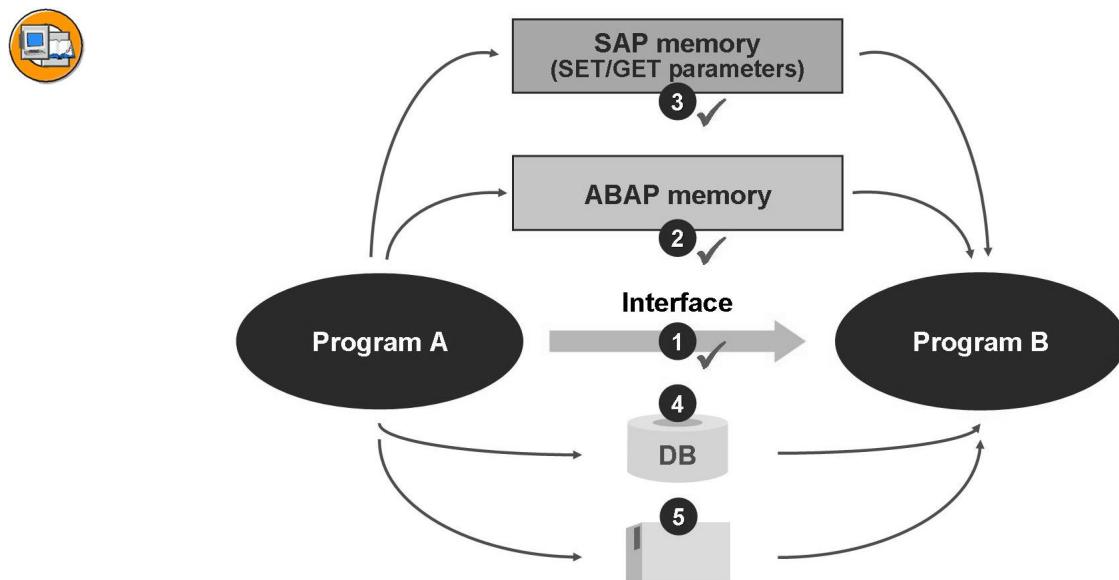


Figure 48: Data Transfer Between Programs: Overview

You can use the following techniques (among others) for passing data:

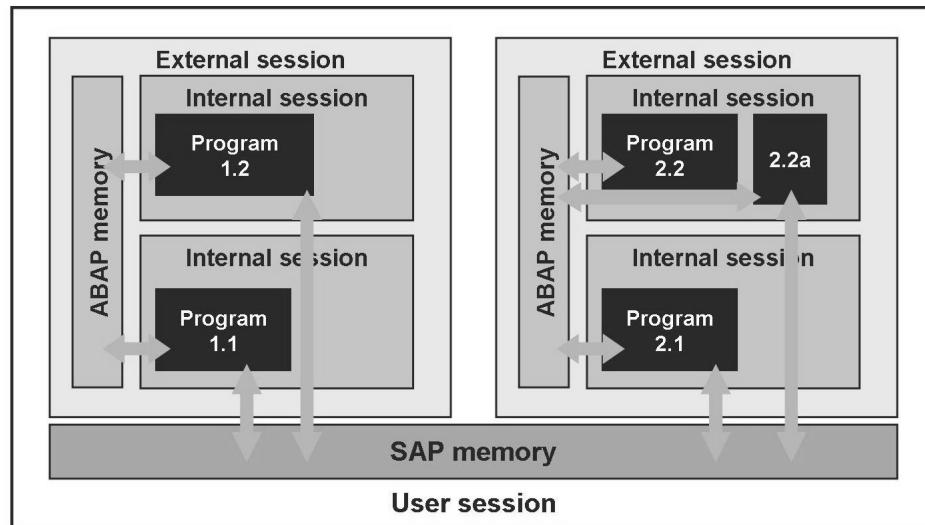
- Interface of the called program
- ABAP memory
- SAP memory
- Database tables
- Local files on the presentation server

We have already discussed transferring data via the program interface. In this section, we introduce data exchange using ABAP memory and SAP memory. For more information about passing data using database tables or the shared buffer, see the ABAP keyword documentation for the **EXPORT** and **IMPORT** statements.

For more information about transferring data between an ABAP program and your presentation server, see the documentation of function modules **WS\_UPLOAD** and **WS\_DOWNLOAD**. You can also use the methods of class **CL\_GUI\_FRONTEND\_SERVICES** for this purpose.

### **ABAP Memory and SAP Memory**

It is often necessary to transfer data between two different programs. However, it is not always possible to specify this data as an addition in the program call. In such cases, you can use the SAP memory and the ABAP memory to transfer data between programs:

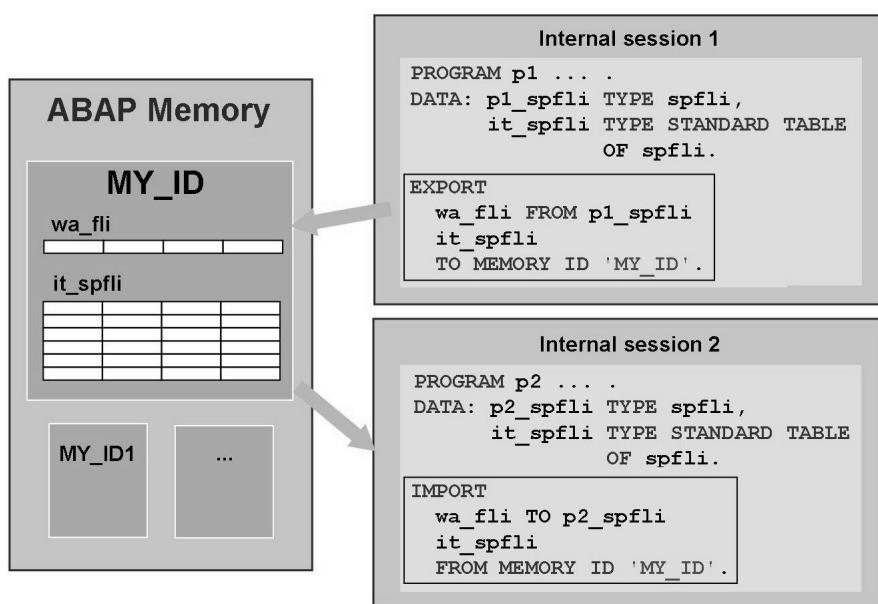


**Figure 49: Range of ABAP Memory and SAP Memory**

- SAP memory is a **user-specific** memory area for storing **field values**. It is therefore not useful for passing data between internal sessions.  
Values in SAP memory are retained for the duration of the user's terminal session. SAP memory can be used between sessions in the same terminal session. **All external sessions** of a user session can access SAP memory.  
You can use the contents of SAP memory as default values for screen fields.
- ABAP memory** is also user-specific. There is a local ABAP memory for each external session. You can use it to exchange any ABAP data objects (fields, structures, internal tables, complex objects) between the internal sessions in any one external session.  
When the user exits an external session (*/i* in the command field), the corresponding ABAP memory is automatically initialized or released.

## ABAP Memory

The EXPORT ... TO MEMORY statement copies any number of ABAP data objects with their current values (data cluster) to the ABAP memory. The **ID** addition (maximum 60 characters long) enables you to identify different clusters.



**Figure 50: Passing Data Using the ABAP Memory**

If you use a new EXPORT TO MEMORY statement for an existing data cluster, the new one overwrites the old one. The IMPORT ... FROM MEMORY ID ... statement allows you to copy data from the ABAP memory into the corresponding fields of your ABAP program.

You can also restrict the selection to a part of the data cluster in the IMPORT statement. The variables into which you want to read data from the cluster in ABAP memory must have the same types in both the exporting and the importing programs.

You use the FREE MEMORY ID ... statement to release a data cluster explicitly.



**Caution:** When you call programs using transaction codes, you can only use the ABAP memory to pass data to the transaction in the insert case (CALL TRANSACTION).

## SAP Memory

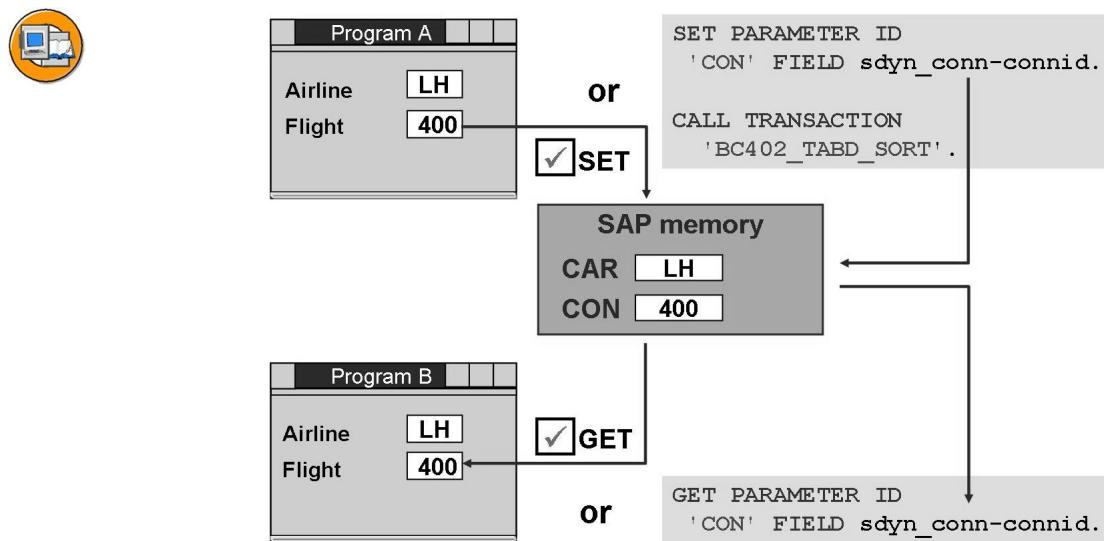


Figure 51: Passing Parameters Using SAP Memory

You can define memory areas (parameters) in SAP memory in various ways:

- By creating input/output fields with reference to the *ABAP Dictionary*. These take the parameter names of the data elements to which they refer.

Alternatively, you can enter a name in the attributes of the input/output fields. Then, you can also choose whether the entries from the field should be transferred to the parameter (SET), or whether the input field should be filled with the value from the parameter (GET).

To find out about the names of the parameters assigned to input fields, display the field help for the field with the (F1) key and choose Technical info.

- You can also fill the memory areas directly with the `SET PARAMETER ID 'PAR_ID' FIELD var.` statement and read from them with the `GET PARAMETER ID 'PAR_ID' FIELD var.` statement.
- Finally, you can define parameters in the *Object Navigator* and let the user fill them with values.

## Memory Management for Dynamic Data Objects

Strings and internal tables are **dynamic** data objects whose memory requirements can change during the program runtime. They are called **deep data objects**, whose working data is managed internally, using references.

→ **Note:** Other deep data objects include references and structures that contain at least one deep component, regardless of their nesting.

Memory management handles deep data objects much differently than **flat data objects**.

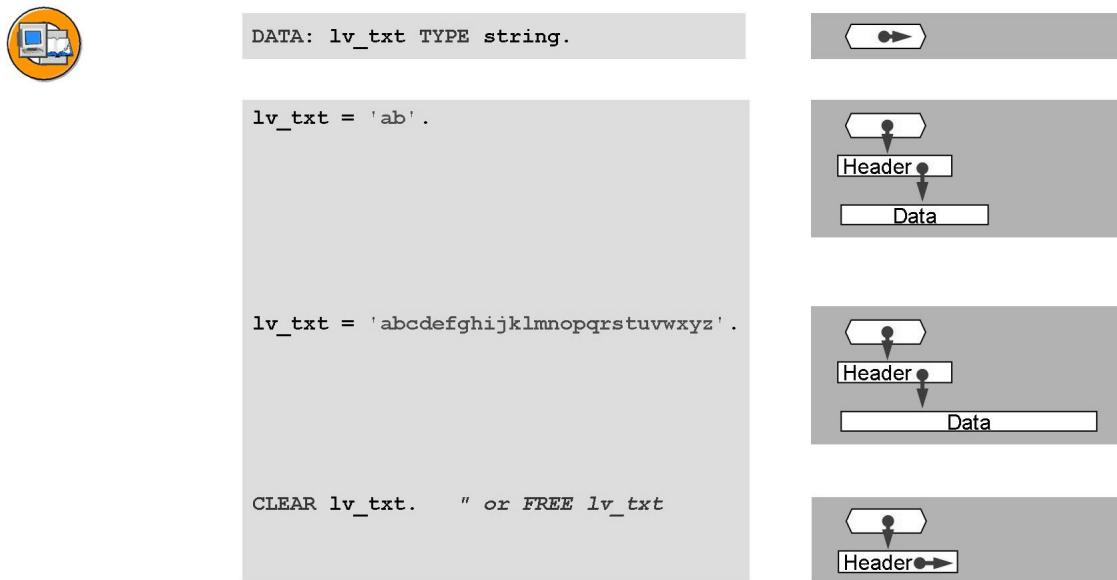
### Memory Management for Flat Data Objects



**Figure 52: Memory Management for Flat Data Objects**

Flat data objects are all elementary data types with fixed lengths and structures that only have flat components. Memory for flat data objects is reserved completely as soon as the program is loaded into the internal session, and is not released until the internal session is deleted. The content of flat data objects represents the actual working data.

## Memory Management for Strings



**Figure 53: Memory Management for Strings**

When a program is loaded, only a reference is created for the statically declared deep data objects at first, such as strings and internal tables. The static memory requirement for this reference is 8 bytes.

When the deep data object is used, additional working memory is requested at the program runtime. For dynamic data objects (strings and internal tables), this memory is for the administration information (header) and the data object itself. The memory requirement for a header depends on the hardware platform, but is around 100 bytes.

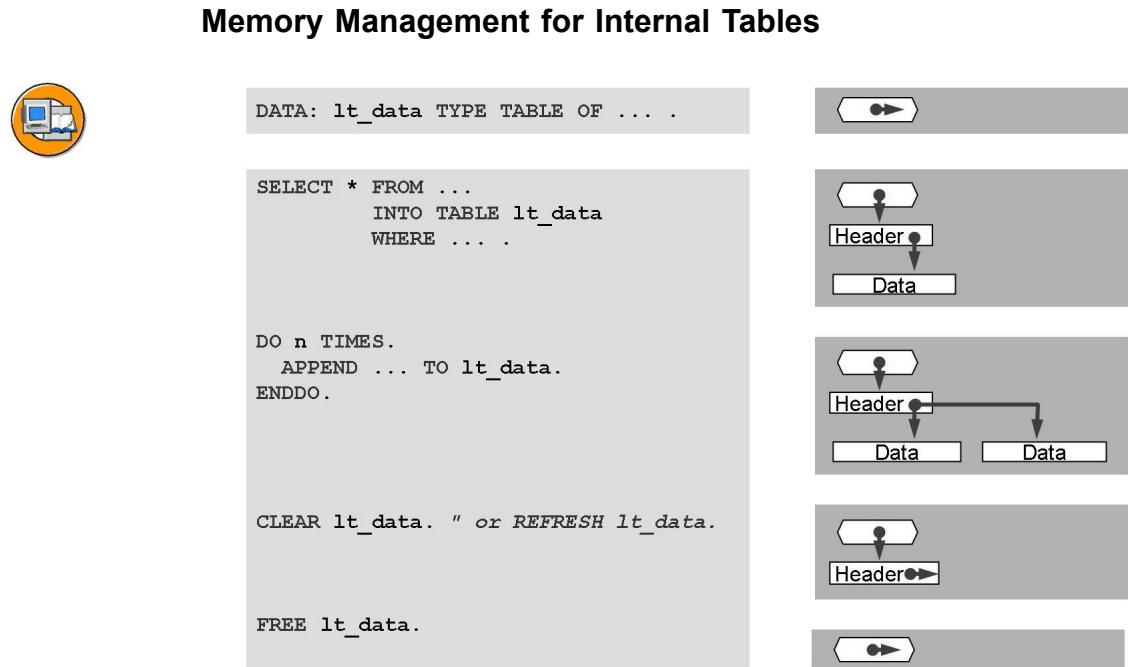
If the requested (allocated) memory is not sufficient during a later access, additional memory can be reserved.

The system allocates more memory than is needed for the current content of the data object, to avoid having to constantly request more as the content slowly grows. The size of strings is limited to the largest contiguous block of memory that can be requested. This is a maximum of 2 GB, but is usually restricted further with profile parameter ztta/max\_memreq\_MB.



**Caution:** No matter what, the requested memory must be available in the current internal mode - otherwise a runtime error occurs.

When a string is initialized with CLEAR or FREE, the data itself is deleted, but the reference variables and the header remain. They are used again during the next memory request. Accordingly, when a string has been used and then deleted, its memory requirement consists of the reference and the requirement of the header.



**Figure 54: Memory Management for Internal Tables**

Memory management for internal tables is similar to management for strings, however, there are a number of peculiarities:

Headers of internal tables normally require less space than other headers, because their data can be distributed among several blocks, which means the header may have to manage references to these blocks.

The CLEAR, REFRESH, and FREE statements have different effects: CLEAR and REFRESH only delete the data itself, while FREE can also delete the table header if it takes up too much memory.

You can influence the size of the memory that is allocated during the initial access when you define the internal table (optional addition INITIAL SIZE).

You can specify a number of rows, n, as numeric literals or numeric constants in the **INITIAL SIZE** addition. The system then selects the size of the first block in memory to fit this exact number of lines.



**Hint:** For global table types in the *ABAP Dictionary*, the *Initial Number of Lines* input field on the *Initialization and Access* tab performs this function.

If the addition is not specified, the number 0 is specified, or the value of n exceeds a maximum value, the system automatically assigns a suitable memory area during the first access. If the number of lines exceeds the specified value, n, during the initial access or later, the system allocates additional memory as usual. Therefore, n is not a fixed upper limit for the capacity of the internal table.



**Caution:** If the specified initial memory requirement is vastly exceeded when the internal table is filled, this might cause the system to allocate much more memory than is needed. Therefore, we only recommend specifying the initial memory requirement if you know the number of entries in the table beforehand and you want to size the initial main memory requirement as specifically as possible.



**Hint:** Specifying an initial memory requirement can make sense when the internal tables themselves are components of table types (nested tables) and the inner tables only contain a few lines.

### Memory Optimization when Copying Dynamic Data Objects

When values are assigned between strings and between internal tables, they are **shared** to save runtime and memory.

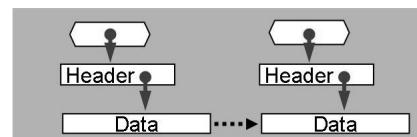
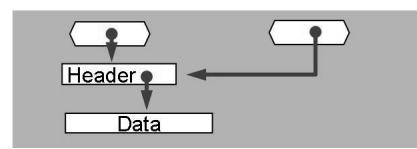
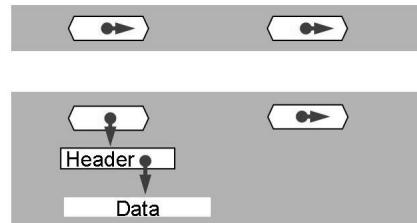


```
DATA: lv_txt1 TYPE string,  
      lv_txt2 TYPE string.
```

```
lv_txt1 = 'abcdef'.
```

```
lv_txt2 = lv_txt1.
```

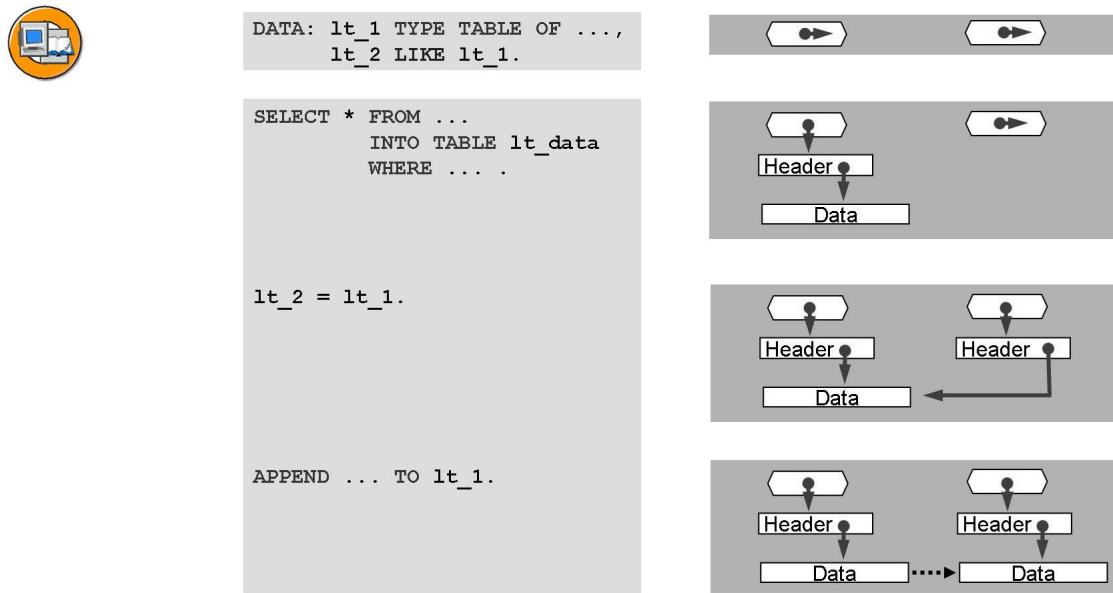
```
CONCATENATE lv_txt1  
            'ghi'  
        INTO lv_txt1.
```



**Figure 55: Sharing Between Strings**

Sharing means the actual data values are not copied at first during an assignment. Instead, only the necessary management entries are copied, so the target object refers to the same data as the source object. When strings are involved, the second internal reference is filled to point to the existing string header. Sharing is

canceled when either the source or the target object is accessed in change mode. This approach is called **copy-on-write semantics**. In this case, the data values are actually copied and the references and header are changed accordingly.



**Figure 56: Sharing Between Internal Tables**

When internal tables are shared, only the actual data is shared. Each table is assigned a separate table header, which refers to the existing table body.

Tables whose line types contain other table types are not shared.



**Hint:** Sharing is also used to **pass values to procedures**. Therefore, there is no major difference with regard to runtime requirements between passing values and passing references for strings and internal tables. The more runtime-intensive copy operation when values are passed takes place when the formal parameters are accessed in change mode within the procedure. In contrast, when references are passed, the actual and formal parameters continue to use the same memory.

### Boxed Components (New in SAP NetWeaver 7.0 EhP2)

As seen above, large static data objects can use a lot of memory – even when they are initial. If we have many identical unused static data objects in our program, this can lead to a significant waste of memory. One example could be an instance attribute with a static structure which is initial in most of the classes instances. Another example could be an internal table where a large number of columns is not used for many table lines.

As of *SAP NetWeaver 7.0 EhP2* **boxed components** may be used to reduce the memory consumption of static data objects. For the time being, boxed components can be:



- Substructures of nested structures
- Structured attributes of classes and interfaces

These are called **static boxes**.

A static box is declared with the **BOXED** addition when defining substructures using **TYPES** or when declaring structured attributes of classes or interfaces using **DATA**. For global structure types, classes and interfaces a new value for the *Typing Option* is available.



**Hint:** It is not possible to use the **BOXED** addition in **DATA** statements that lie outside of class definitions. To define boxed components outside of class definitions, you have to use the **TYPES** statement or to define a suitable global structure type.

The following figure gives you an example of a nested structure with a boxed static substructure:



```

TYPES: BEGIN OF ty_s_sub,
        subc1 TYPE i,
        subc2 TYPE c LENGTH 4,
    END OF ty_s_sub.

TYPES: BEGIN OF ty_s_boxed,
        comp1 TYPE c LENGTH 4,
        sub   TYPE ty_s_sub BOXED,
        comp2 TYPE i,
    END OF ty_s_boxed.

DATA gs_boxed TYPE ty_s_boxed.

gs_boxed-sub-subc1 = 'abcd'.

CLEAR gs_boxed.

```

Initial Value Sharing:  
Reference points to  
initial structure in PXA

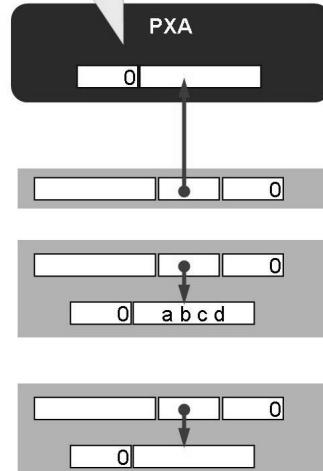


Figure 57: Structure with Boxed Component

The content of the boxed subcomponent SUB is not stored inside of structure GS\_BOXED, directly. Instead, the subcomponent is internally represented by a reference. As long as the subcomponent is still initial, this reference points to an appropriately typed initial structure stored centrally in the PXA (Program eXecution Area). This is called **Initial Value Sharing** for boxed components.

The initial value sharing is **revoked** if, for example, a write access to the substructure takes place, after which the substructure is not initial any more. As illustrated in the above figure, the substructure is dynamically created within the internal session of the program and the reference is changed to point there.



**Hint:** Once the initial value sharing is revoked, it remains like that. The CLEAR or FREE statement only initializes the components of the substructure but does not release the memory allocated for the substructure.

Operations that revoke the initial value sharing of a boxed component are:

- Write access to the static box or one of its components
- Using a static box or one of its components as an actual parameter for procedure calls
- Addressing the static boxed component or one of its subcomponents with a data reference or a field symbol.



**Hint:** A structure that contains a boxed component is always deep, even if all its components (including the boxed subcomponent) are of static length.

The following figure gives you an example for a boxed instance attribute of a class:



```

TYPES: BEGIN OF ty_s_struct,
        comp1 TYPE i,
        comp2 TYPE c LENGTH 4,
    END OF ty_s_struct.

CLASS lcl_boxed DEFINITION.
  PUBLIC SECTION.
  DATA:
    attr TYPE ty_s_struct BOXED.
ENDCLASS.

DATA go_boxed TYPE REF TO lcl_boxed.

```

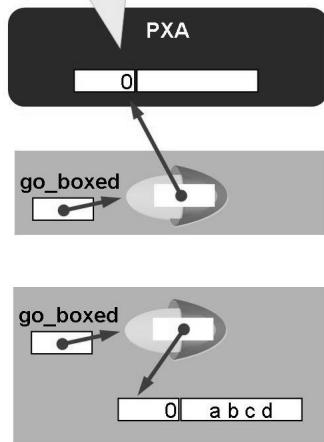
```

CREATE OBJECT go_boxed.

go_boxed->attr-comp2 = 'abcd'.

```

**Sharing of initial value:**  
Reference points to  
initial structure in PXA



**Figure 58: Instance with a Boxed Attribute**

If the boxed attribute remains initial in many instances of the class, all instances share the same initial value. This can significantly reduce the memory consumption of the program.

Similarly, the overall size of an internal table can be reduced if a structure type having a boxed component is used as line type. This is illustrated in the following figure.



```

TYPES: BEGIN OF ty_s_sub,
      subc1 TYPE i,
      subc2 TYPE c LENGTH 4,
   END OF ty_s_sub.

TYPES: BEGIN OF ty_s_boxed,
      comp1 TYPE c LENGTH 4,
      sub   TYPE ty_s_sub BOXED,
      comp2 TYPE i,
   END OF ty_s_boxed.

DATA:
      gs_boxed TYPE ty_s_boxed.
      gt_boxed TYPE TABLE OF ty_s_boxed.

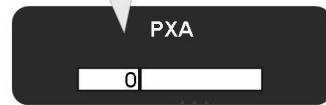
```

```

DO 3 TIMES.
  gs_boxed-comp1 = 'abcd'.
  gs_boxed-comp2 = sy-index.
  APPEND gs_boxed TO gt_boxed.
ENDDO.

```

**Sharing of initial value:**  
all References point to  
same initial structure in PXA



gt_boxed				
a	b	c	d	1
a	b	c	d	2
a	b	c	d	3

**Figure 59: Example: Internal Tables and Boxed Components**

## When to Use Boxed Components?

You only should use boxed components if the following requirements are fulfilled:

### Boxed component often remains initial

The advantage of boxed components lies in the reduced memory usage through initial value sharing. That implies that boxed components have no advantage if the boxed component never or hardly ever remains initial.

### Boxed component is not too small

If the substructure or structured attribute is too small the memory savings can not counterbalance the 8 Byte overhead for the reference.

### Boxed component is used often

The memory savings through boxed components scale with the number of times the initial structure is actually shared.



**Hint:** This does not necessarily mean that the class or nested structure has to have several instances within the same execution of the program. As the PXA exists only once per server instance we can also benefit from boxed components if the program is executed many times on the same server instance.

## Exercise 4: ABAP Memory and SAP Memory

### Exercise Objectives

After completing this exercise, you will be able to:

- Use the ABAP memory to exchange data between programs
- Use the SAP memory to preassign fields in a transaction call

### Business Example

An existing program selects data based on user entries and outputs it in a list. You call this program from within another program. But instead of displaying the data directly with the called report, you want to load the selected data into the SAP memory and process it further in the calling program. You also want to use parameters in the SAP memory to preassign fields when calling a transaction.

#### Template:

BC402\_INS\_FLIGHT\_LIST (called program)  
BC402\_PCS\_CONN\_LIST (calling program)

#### Solution:

BC402\_MMS\_FLIGHT\_LIST (called program)  
BC402\_MMS\_CONN\_LIST (calling program)

### Task 1:

Copy the **called** program, **BC402\_INS\_FLIGHT\_LIST** (or your own program, **ZBC402\_##\_FLIGHT\_LIST**), to the name **ZBC402\_##\_FLIGHT\_LIST\_MEM**. Enhance the program so that it can selectively output the flights in a list or save the internal table with its data in the ABAP memory. Create an invisible parameter on the selection screen that enables storage in the ABAP memory when the program is called with SUBMIT.

1. Copy the program and all its subcomponents.
2. Define an additional parameter for the selection screen (suggested name: **pa\_list**). Type the parameter with type ABAP\_BOOL from the ABAP type group, for example.
3. Check the documentation to find a suitable addition for the PARAMETERS statement to suppress the parameter when displaying the selection screen.

*Continued on next page*

4. Supplement the program so it suppresses the list output when the parameter is not set to its default value. In this case, you want to export the read data into the ABAP memory. Choose a suitable ID for your data cluster (suggested name: **BC402\_##\_FLIGHTS**).

## Task 2:

Copy the **calling** program, **BC402\_PCS\_CONN\_LIST** (or your own program, **ZBC402\_##\_CONN\_LIST**), to the name **ZBC402\_##\_CONN\_LIST\_MEM** and modify the name of the called program in the SUBMIT statement. Call the program so it saves the data in the ABAP memory instead of outputting it directly.

1. Copy the calling program.
2. Modify the SUBMIT statement to call program, **ZBC402\_##\_FLIGHT\_LIST\_MEM**. Supply the new parameter with a value to suppress the list display and write the data to the ABAP memory.

## Task 3:

After the program call, read the data from the ABAP memory and use function module BC402\_DISPLAY\_TABLE to display the data in an ALV grid. Do not forget to delete the data in the ABAP memory after it is read.

1. After the program call, import the data from the ABAP memory. To do so, create a suitably typed internal table. If an error occurs, respond with error message 361 from message class BC402.
2. Implement a call of function module BC402\_DISPLAY\_TABLE. Pass on the data from the ABAP memory to this module for display.
3. Optional: Deal with the (class-based) exception of the function module.
4. Release the memory for your cluster to delete the data from the ABAP memory.

## Task 4:

Analyze transaction BC402MCAR. Is it possible to preassign the input field on the first screen using a parameter in the SAP memory? If so, use this option. Remove the USING and MODE additions from the CALL TRANSACTION statement and suppress the display of the first screen.

1. Find out whether the input field from screen 0100 in program **SAPMBC402\_IND\_CARRIER** is linked with a parameter ID. If so, what is its name?
2. Check whether automatic preassignment from the SAP memory is active for the field.

*Continued on next page*

3. Use a suitable statement to set the value of the parameter to the desired value in the SAP memory before the transaction call.
4. Remove the USING and MODE additions from the CALL TRANSACTION statement. Instead, use a suitable addition to suppress the first screen.

## Solution 4: ABAP Memory and SAP Memory

### Task 1:

Copy the **called** program, **BC402\_INS\_FLIGHT\_LIST** (or your own program, **ZBC402\_##\_FLIGHT\_LIST**), to the name **ZBC402\_##\_FLIGHT\_LIST\_MEM**. Enhance the program so that it can selectively output the flights in a list or save the internal table with its data in the ABAP memory. Create an invisible parameter on the selection screen that enables storage in the ABAP memory when the program is called with SUBMIT.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Define an additional parameter for the selection screen (suggested name: **pa\_list**). Type the parameter with type ABAP\_BOOL from the ABAP type group, for example.
  - a) See the source code excerpt from the model solution.
3. Check the documentation to find a suitable addition for the PARAMETERS statement to suppress the parameter when displaying the selection screen.
  - a) NO-DISPLAY addition; see source code excerpt from the model solution.
4. Supplement the program so it suppresses the list output when the parameter is not set to its default value. In this case, you want to export the read data into the ABAP memory. Choose a suitable ID for your data cluster (suggested name: **BC402\_##\_FLIGHTS**).
  - a) See the source code excerpt from the model solution.

### Task 2:

Copy the **calling** program, **BC402\_PCS\_CONN\_LIST** (or your own program, **ZBC402\_##\_CONN\_LIST**), to the name **ZBC402\_##\_CONN\_LIST\_MEM** and modify the name of the called program in the SUBMIT statement. Call the program so it saves the data in the ABAP memory instead of outputting it directly.

1. Copy the calling program.
  - a) Carry out this step in the usual manner.
2. Modify the SUBMIT statement to call program, **ZBC402\_##\_FLIGHT\_LIST\_MEM**. Supply the new parameter with a value to suppress the list display and write the data to the ABAP memory.
  - a)

*Continued on next page*

### Task 3:

After the program call, read the data from the ABAP memory and use function module BC402\_DISPLAY\_TABLE to display the data in an ALV grid. Do not forget to delete the data in the ABAP memory after it is read.

1. After the program call, import the data from the ABAP memory. To do so, create a suitably typed internal table. If an error occurs, respond with error message 361 from message class BC402.
  - a) See the source code excerpt from the model solution.
2. Implement a call of function module BC402\_DISPLAY\_TABLE. Pass on the data from the ABAP memory to this module for display.
  - a) See the source code excerpt from the model solution.
3. Optional: Deal with the (class-based) exception of the function module.
  - a)
4. Release the memory for your cluster to delete the data from the ABAP memory.
  - a) See the source code excerpt from the model solution.

### Task 4:

Analyze transaction BC402MCAR. Is it possible to preassign the input field on the first screen using a parameter in the SAP memory? If so, use this option. Remove the USING and MODE additions from the CALL TRANSACTION statement and suppress the display of the first screen.

1. Find out whether the input field from screen 0100 in program SAPMBC402\_IND\_CARRIER is linked with a parameter ID. If so, what is its name?
  - a) Start the transaction.
  - b) Call up the help (F1 key) for the input field and then the technical information.
  - c) The *Parameter ID* has the value *CAR*.
2. Check whether automatic preassignment from the SAP memory is active for the field.
  - a) Navigate to screen 0100.
  - b) Analyze the attributes for the input screen (either in the list of elements or using the attributes window in the graphical layout editor).
  - c) The *Get Parameters* checkbox is set.

*Continued on next page*

3. Use a suitable statement to set the value of the parameter to the desired value in the SAP memory before the transaction call.
  - a) See the source code excerpt from the model solution.
4. Remove the USING and MODE additions from the CALL TRANSACTION statement. Instead, use a suitable addition to suppress the first screen.
  - a) AND SKIP FIRST SCREEN addition; see source code excerpt from the model solution.

## Result

Source code excerpt from the model solution:

### **BC402\_MMS\_FLIGHT\_LIST (called program)**

```

REPORT  bc402_mms_flight_list MESSAGE-ID bc402.

TYPE-POOLS: abap.

DATA: gt_flight TYPE TABLE OF sflight,
      gs_flight          sflight.

PARAMETERS pa_car TYPE sflight-carrid.
SELECT-OPTIONS so_con FOR gs_flight-connid.

PARAMETERS pa_list TYPE abap_bool DEFAULT abap_true NO-DISPLAY.

START-OF-SELECTION.

      SELECT * FROM sflight INTO TABLE gt_flight
      WHERE carrid =  pa_car AND
            connid IN so_con.

      IF sy-subrc <> 0.
      MESSAGE e038.
      ENDIF.

      IF pa_list = abap_true.

      LOOP AT gt_flight INTO gs_flight.

      WRITE: /
            gs_flight-carrid,

```

*Continued on next page*

```

        gs_flight-connid,
        gs_flight-fldate,
        gs_flight-price      CURRENCY gs_flight-currency,
        gs_flight-currency,
        gs_flight-planetype,
        gs_flight-seatsmax,
        gs_flight-seatsocc.

ENDLOOP.

ELSE.

EXPORT flights FROM gt_flight
    TO MEMORY ID 'BC402_FLIGHTS'.

ENDIF.

```

**BC402\_MMS\_CONN\_LIST (calling program),  
Method: on\_double\_click**

```

METHOD on_double_click.

DATA lt_sflight TYPE TABLE OF sflight.

DATA ls_conn LIKE LINE OF gt_conn.

READ TABLE gt_conn INTO ls_conn INDEX row.

CASE column.
    WHEN 'CARRID'.

        SET PARAMETER ID 'CAR' FIELD ls_conn-carrid.

        CALL TRANSACTION 'BC402MCAR'
            AND SKIP FIRST SCREEN.

    WHEN OTHERS.

        SUBMIT bc402_mms_flight_list
            AND RETURN
            WITH pa_car EQ ls_conn-carrid

```

*Continued on next page*

```
WITH so_con EQ ls_conn-connid  
  
WITH pa_list EQ space.  
  
IMPORT flights TO lt_sflight  
FROM MEMORY ID 'BC402_FLIGHTS'.  
  
IF sy-subrc <> 0.  
MESSAGE e361.  
ELSE.  
CALL FUNCTION 'BC402_DISPLAY_TABLE'  
CHANGING  
ct_table = lt_sflight.  
ENDIF.  
  
FREE MEMORY ID 'BC402_FLIGHTS'.  
  
ENDCASE.  
  
ENDMETHOD.          "on_double_click
```



## Lesson Summary

You should now be able to:

- Describe how the ABAP runtime environment executes programs
- Describe the importance of the runtime object
- Explain the terms "roll area" and "PXA"
- Explain memory management in the ABAP runtime environment
- Describe how dynamic data objects are managed
- Explain how "Boxed Components" are managed and when they should be used

# Lesson: Shared Objects

## Lesson Overview

Starting in *SAP NetWeaver 2004*, you can save data as shared objects in shared memory. In this lesson, we examine this concept and its implementation.

The advantage of this technique is that data can be saved as objects. Programs can access this data quickly, independently of the user session. This technique was developed with the primary objective of saving catalog information and shopping carts. An attribute of catalogs is that they are read often, but only changed rarely. In the case of the shopping cart, the data is written by one application and then read by another program at a different time (in a different user session). Here as well, this technique delivers considerably better performance than saving the data persistently in the database.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program

## Business Example

You want to develop a transaction in which many users can access the same data concurrently without having to read it from the database each time. You know that shared objects can be used to make data in the main memory accessible across session boundaries. Accordingly, you familiarize yourself with this technique.

## Shared Objects - Introduction

Starting in *SAP NetWeaver 2004*, you can save data as shared objects in shared memory, across different programs and even different user sessions. Accordingly, you can create applications in which users write data to this area. Other users can then read this data later.



We can imagine many different potential uses for shared objects:

- Saving a **catalog**  
An “author” writes the catalog to the shared objects area – many users can then access this catalog at the same time.
- Saving a **shopping cart**  
The buyer fills the shopping cart and the seller reads it later.

## Shared Memory

Shared memory is a memory area on an application server that can be accessed by all the ABAP programs running on that server.

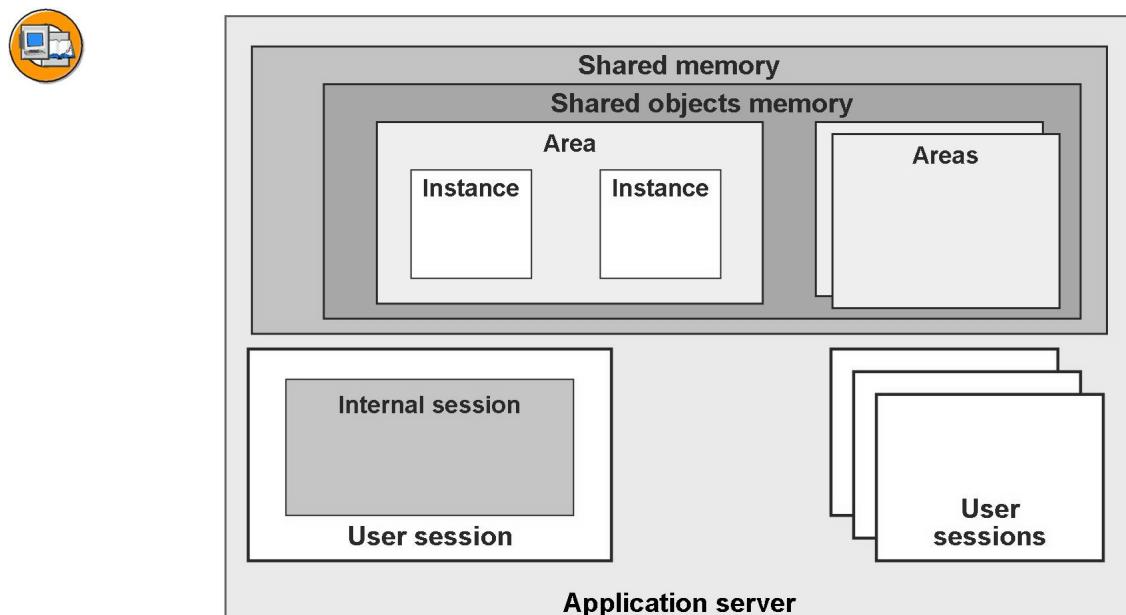
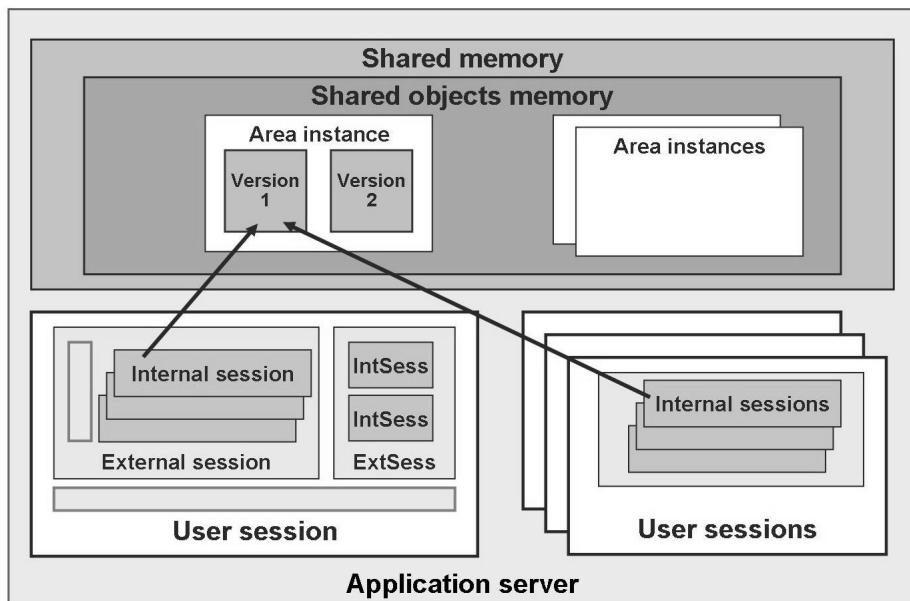


Figure 60: Memory Model of an Application Server

Before shared objects were introduced, ABAP statements had to use the EXPORT and IMPORT statements with the SHARED BUFFER and SHARED MEMORY additions to access this memory area. Instances of classes "lived" exclusively in the internal session of an ABAP program. With the introduction of shared objects in *SAP NetWeaver 2004*, shared memory has been enhanced with **Shared Objects Memory**, where the shared objects can be saved. Shared objects are saved in areas of shared memory.

→ **Note:** At the present time, instances of classes can be saved. It is not (yet) possible to save any data objects as shared objects. However, data objects (aside from reference variables) can be saved as attributes of classes.



**Figure 61: Accessing Shared Objects**

### Properties of Shared Objects



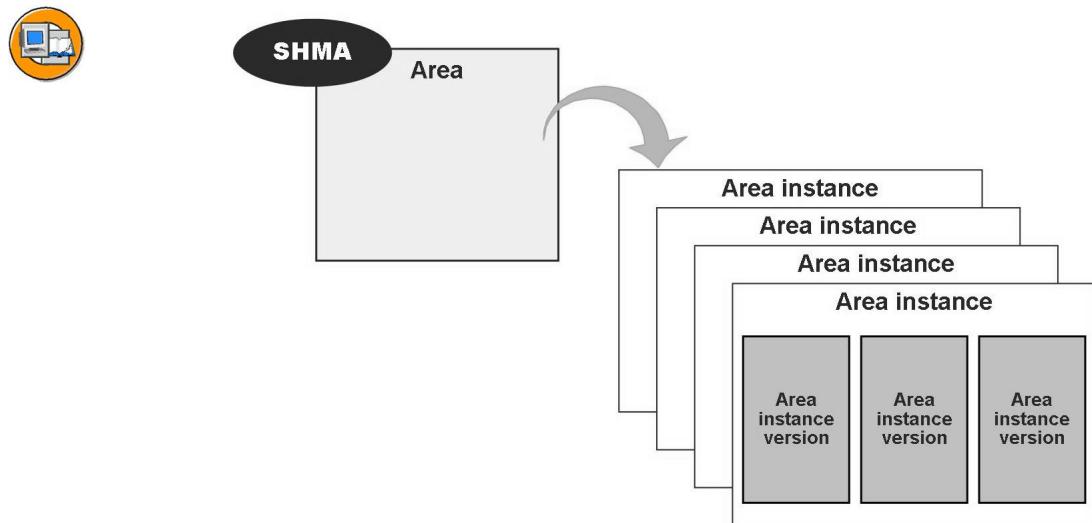
- Cross-program buffering of data that is read often, but rarely written  
(Recommendation: Once per day to once per hour)
- Concurrent read accesses are supported
- Access is controlled by a lock mechanism
- Data is saved as attributes of objects
- Memory bottlenecks result in runtime errors and have to be caught

The write accesses should be seldom because writing data to the shared objects area is performance-intensive. Specifically, you want to optimize runtime, which would be lost if write access were more frequent.

→ **Note:** SAP also uses shared objects in *SAP NetWeaver 2004*. For example, this technique is used to navigate in the *ABAP Workbench*. In addition to saving memory (around 3 MB per user logon), navigation during the first access is faster by up to a factor of 100.

A prerequisite for saving an object in shared memory is that the class of that object is defined with the SHARED MEMORY ENABLED addition of the CLASS statement, or that the *Shared Memory Enabled* attribute is selected in the *Class Builder*.

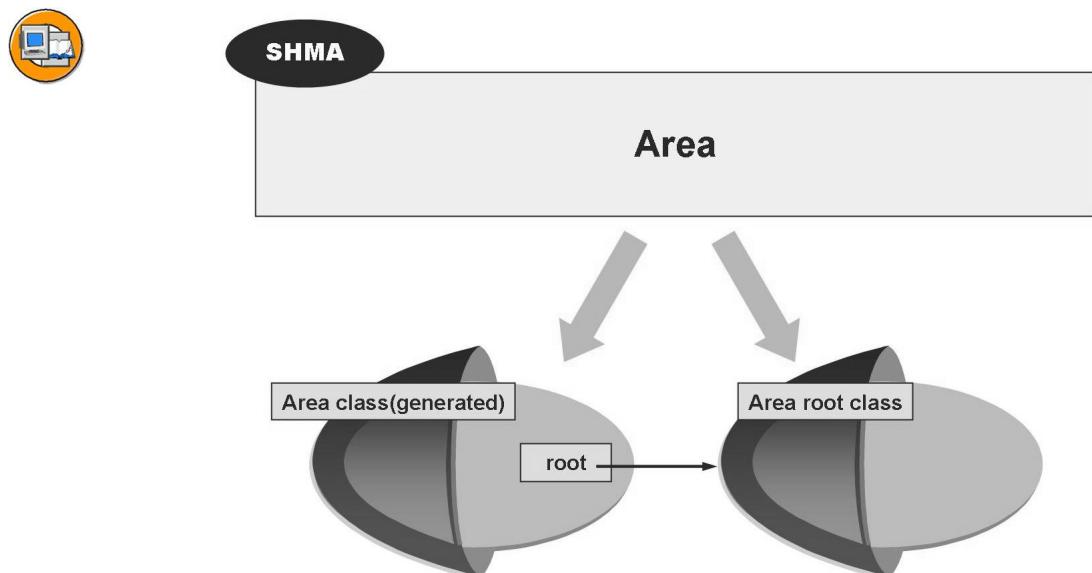
## Areas and Area Instances



**Figure 62: Areas and Area Instances**

An area is the template for area instances in shared memory. One area can spawn several area instances, which differ by name. In addition, an area instance can have several versions (“area instance versions”), which differ in their version IDs. In the simplest case, without version management, an area instance consists of a single area instance version.

## Area Classes and Area Handles



**Figure 63: Creating an Area**

You define an area in transaction SHMA. This creates a global, final area class **of the same name**. This is a subclass of CL\_SHM\_AREA. In an ABAP program, the area is accessed exclusively using methods of the generated area class.



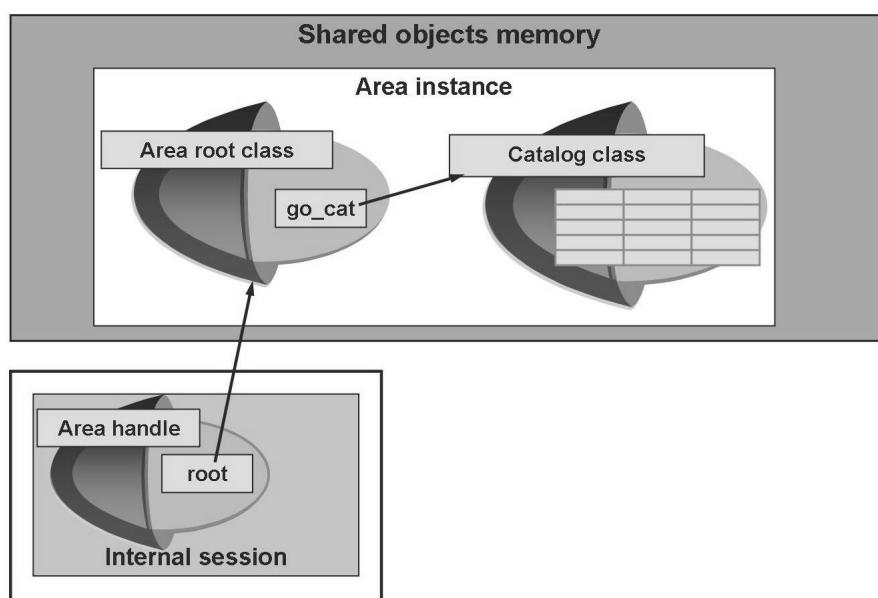
**Hint:** Since the area and area class have the same name, it makes sense to name the area according to the naming conventions for classes – ZCL\_\*, for example, in the customer namespace.

You can use static methods (“Attach” methods) of an area class to attach an ABAP program (or its internal session, where the ABAP program is processed) to an area instance in shared memory. When you attach an ABAP program, an instance of the area class is created as an area handle.

The above diagram also shows another class, which is called the area root class. You can create any number of objects in an area instance, depending on your specific program. You access these objects uniformly through the instance of the area root class.

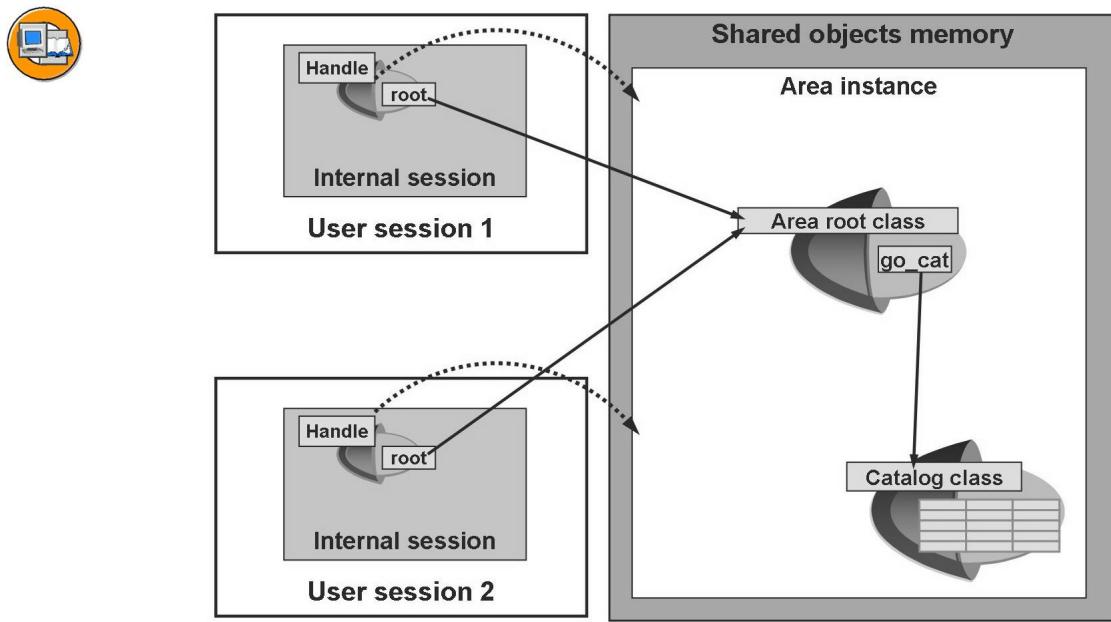
→ **Note:** We limit ourselves to two classes in a specific example.

## Developing an Example Application



**Figure 64: Example Application**

In this lesson, we develop a simple catalog application as an example. We want to use shared objects to create a catalog of flight dates, where the users can select any flight quickly.



**Figure 65: Accessing Areas**

The diagram again illustrates the fact that any programs can access objects in the shared objects memory. In this case, two applications, which run in different user sessions, access objects in the same area. Therefore, we need several things for the example application:

- Creating an area
- Developing a program to create an area instance
- Developing a program to read data from the area

### Creating an Area

Shared objects are saved in areas of shared memory. You use transaction SHMA to create and manage areas and their attributes.



**Figure 66: Area Management**



**Hint:** Note that you have to call transaction code SHMA directly in SAP NetWeaver 2004, because it has not been added to the menu tree yet.

Call transaction code SHMA and enter the name of the area. The usual namespace rules apply, which means the area name has to start with Y or Z in the customer system. Namespaces containing slashes are also supported.



**Hint:** Note that, for this area, an identically named global and final area class is generated as a subclass of CL\_SHM\_AREA. Therefore, it makes sense to choose the area name wisely, as shown in our example.

In an ABAP program, an area is accessed exclusively using methods of the generated area class.



<b>Basic Properties</b>	
Area Name	CL_BC402_SHM_AREA
Description	Demo BC402: Shared Objects Gebiet
Root Class	CL_BC402_SHM_ROOT
<input checked="" type="checkbox"/> Client-Specific Area <input type="checkbox"/> Aut. Area Structuring <input type="checkbox"/> Transactional Area	
<input type="button" value="Attributes"/> <input type="button" value="History"/>	
<b>Fixed Properties</b>	
When Changing:	<input checked="" type="radio"/> All Area Instances Are Invalidated. <input type="radio"/> All Locks Are Released. <input checked="" type="checkbox"/> With Versioning
<b>Dynamic Properties</b>	
When Changing:	<input type="radio"/> Active Area Instances Are Not Invalidated. <input checked="" type="radio"/> All Locks Remain.
Constructor Class	1208200200 Displacement Not Possible
Displacem. Type	1208200200 Displacement Not Possible
<b>Runtime Setting (Default)</b>	
Area Structure	1107197100 No Autostart
Version Size	UNLIMITED No Limit kByte
No. of Versions	VALUE Maximum 3
Lifetime	1125600000 No Entry Minutes

**Figure 67: Maintaining Areas**

After choosing one of the pushbuttons Create, Change or Display, the maintenance screen for areas is displayed.

Each area is linked with a global **area root class**, whose attributes can contain proprietary data and references to other shared memory-enabled classes. You have to assign the area root class to an area when you maintain it. If an area instance version is not empty, it has to contain at least an instance of the area root class as its root object, which is used to reference other objects. When you generate the area class, a *ROOT* attribute is generated and typed with the static type of the area root class.

## Other Important Terms for Creating an Area

### Client-dependent area

Areas, and thus the objects within an area, do not have a client ID by default. You can specify an area as client-dependent, however. In client-dependent areas, the methods of the area class for accessing an area instance refer to the active client by default. You can use the optional importing parameter **CLIENT** to access another client explicitly.

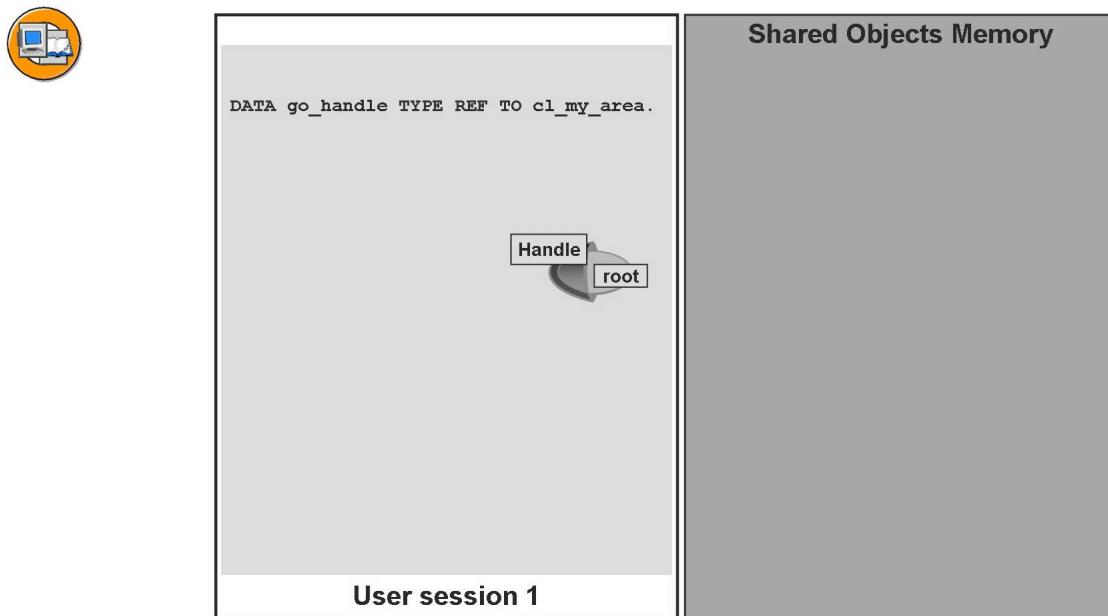
### Transactional area

An area instance version of a transactional area is not active immediately after you remove a change lock with the **DETACH\_COMMIT** method; it is not active until the next database commit.

This is particularly helpful in implementing shopping carts in the shared objects memory.

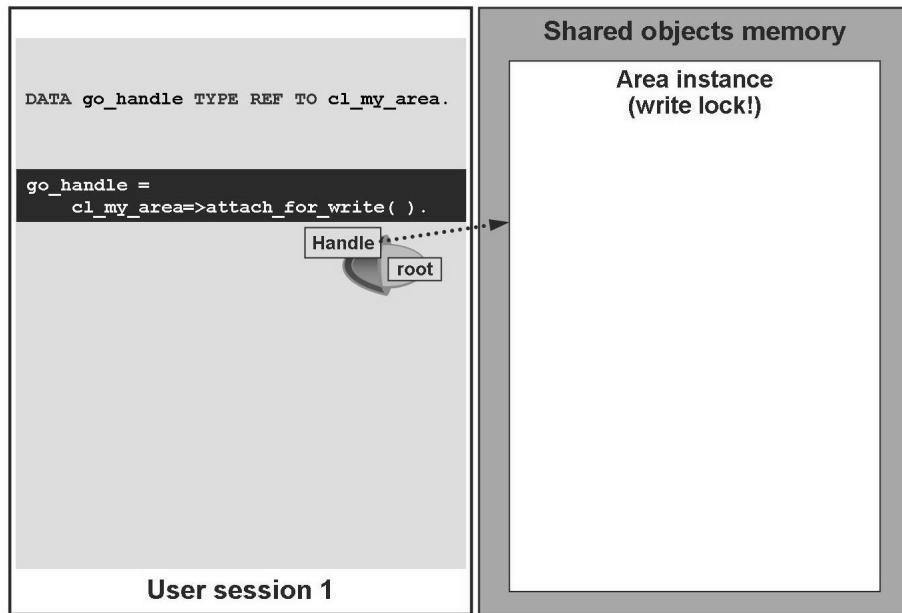
## Setting Up an Area Instance

In the last section, we defined an area. We also created the classes that will be instantiated in this area. In this section, we examine the statements that we can use to create an area **instance**. We continue to use the example introduced above.



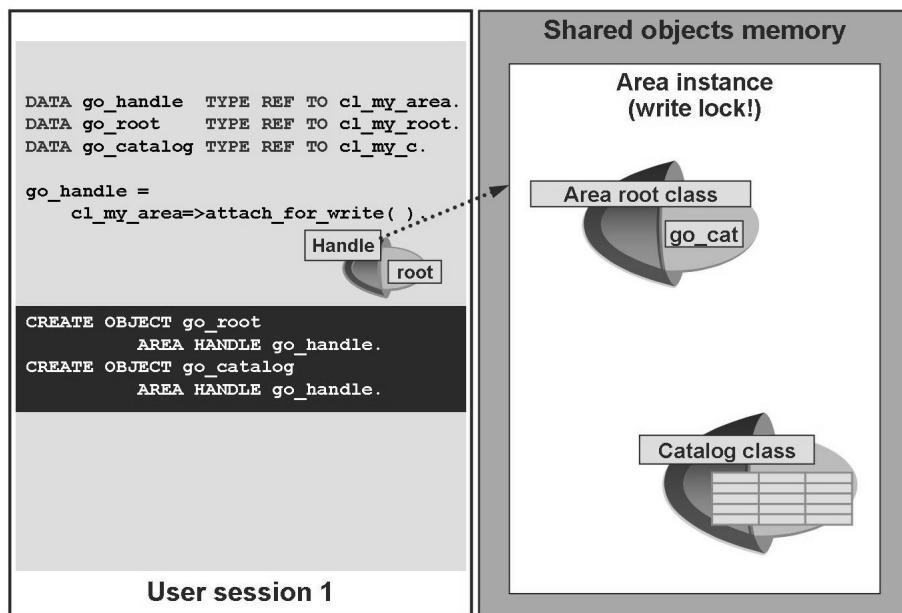
**Figure 68: Before Creating an Area Instance**

When you create an area, a global final class with the same name is also created. To set up an area or access an existing area, you need a reference variable that is typed with the generated area class. This reference serves as a handle for accessing the area.



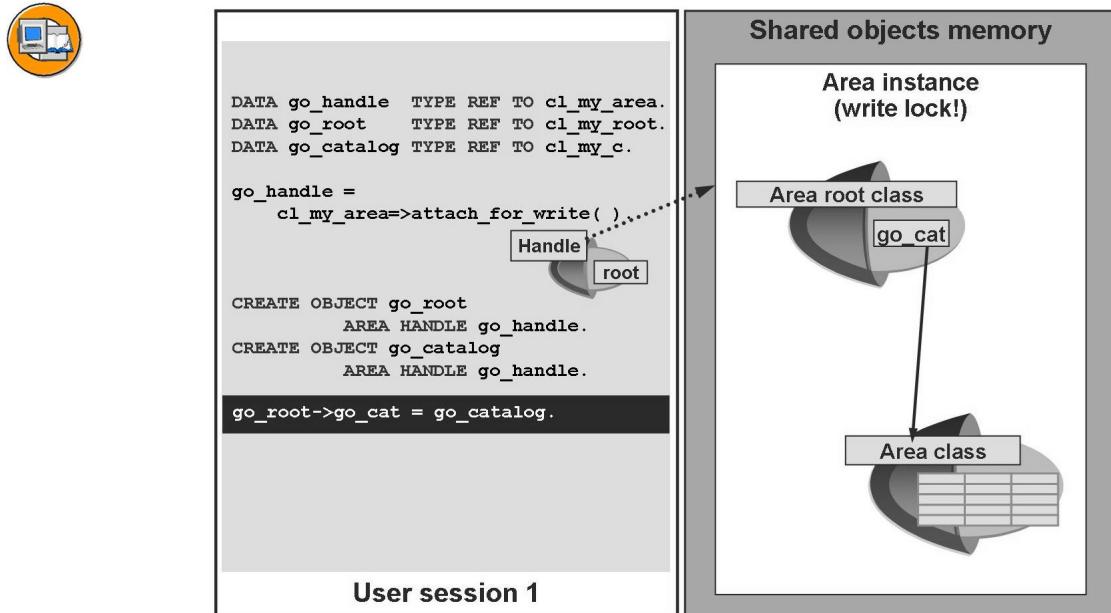
**Figure 69: Creating an Area Instance**

When you instantiate the area class, an instance of the area is created in shared memory. The program has a handle for this instance of the area; all future operations are performed using this handle.



**Figure 70: Generating Objects in Shared Memory**

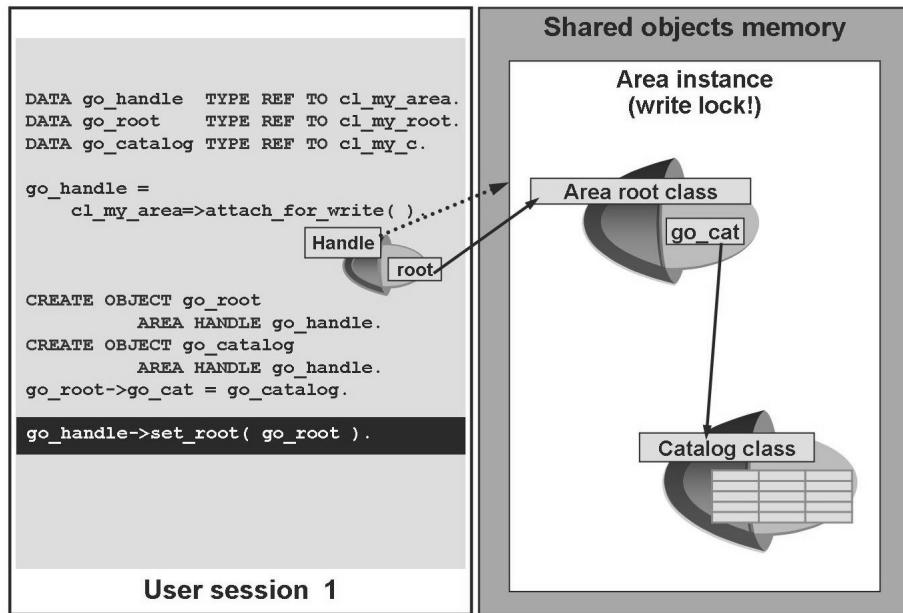
Once the area instance has been created, the objects can be created in the shared objects memory. To do so, use the AREA HANDLE addition for the CREATE OBJECT statement. This tells the system the area instance where the objects shall be created.



**Figure 71: Generating Objects in Shared Memory II**

In the above diagram, both objects are instantiated from within the program. Alternatively, you can instantiate the root object from within the program. The other objects in this area instance can then be created from the constructor of the root object.

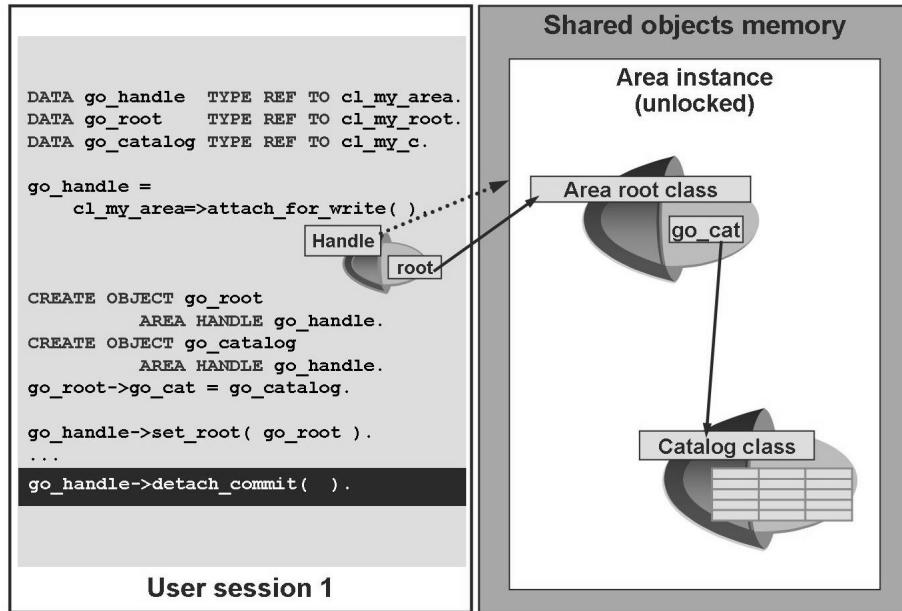
You do not have to assign the references in this case.



**Figure 72: Setting the Root Object**

To be able to address the objects created in the area instance, you have to assign the root object to the ROOT attribute of the area handle. To do so, you use the SET\_ROOT method of the area handle.

As a consequence, any program of any application can access this area. To do so, the application merely has to fetch a reference to the area instance, and can then immediately access the objects contained in that area instance.

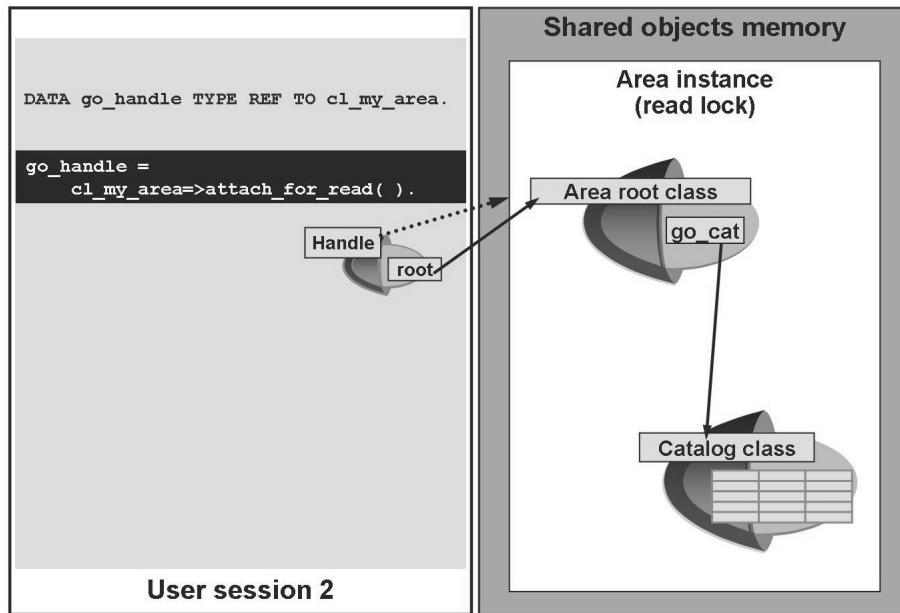


**Figure 73: Releasing the Write Lock**

Read access of an area instance is not possible until the write lock has been released. To do so, use the DETACH\_COMMIT method, which inherits the area class from class CL\_SHM\_AREA.

### Accessing an Area Instance

Once an area instance has been set up, any other users and applications can access it. The reading programs have to implement the following steps:

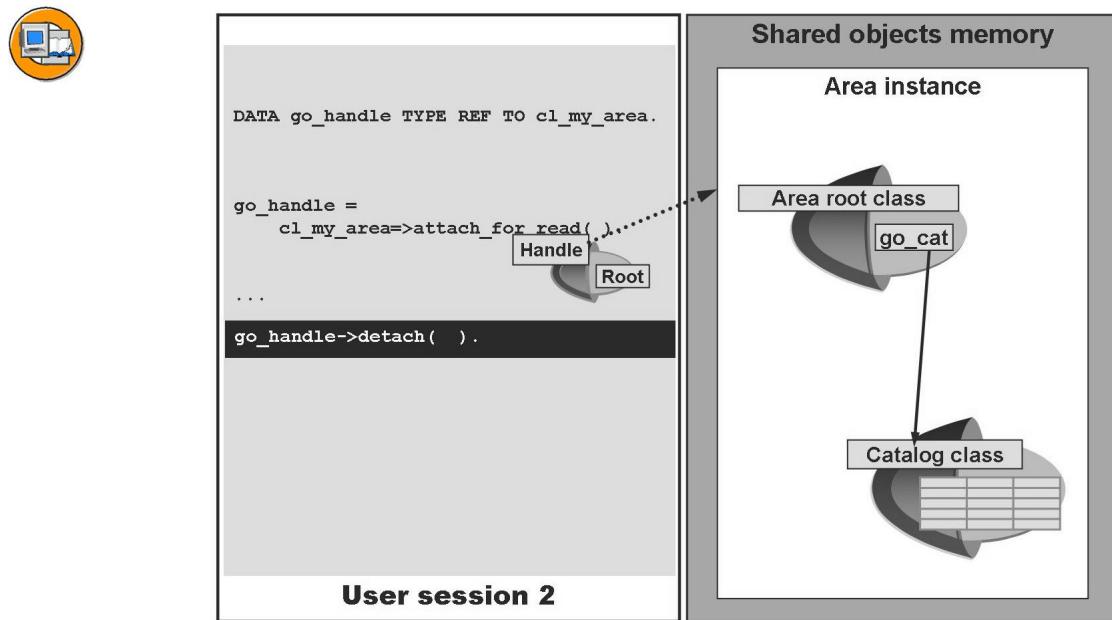


**Figure 74: Accessing an Existing Object in Shared Memory**

The read program first needs a reference variable that is typed with the area class. This reference variable serves as a handle for the area instance that is being accessed.

The program also has to obtain the handle for the area instance. This is performed using method ATTACH\_FOR\_READ, which class CL\_SHM\_AREA provides. This sets a read lock that prevents the area instance from being erased during the access.

The objects in this area instance can now be accessed, always using the area handle.



**Figure 75: Canceling the Read Lock**

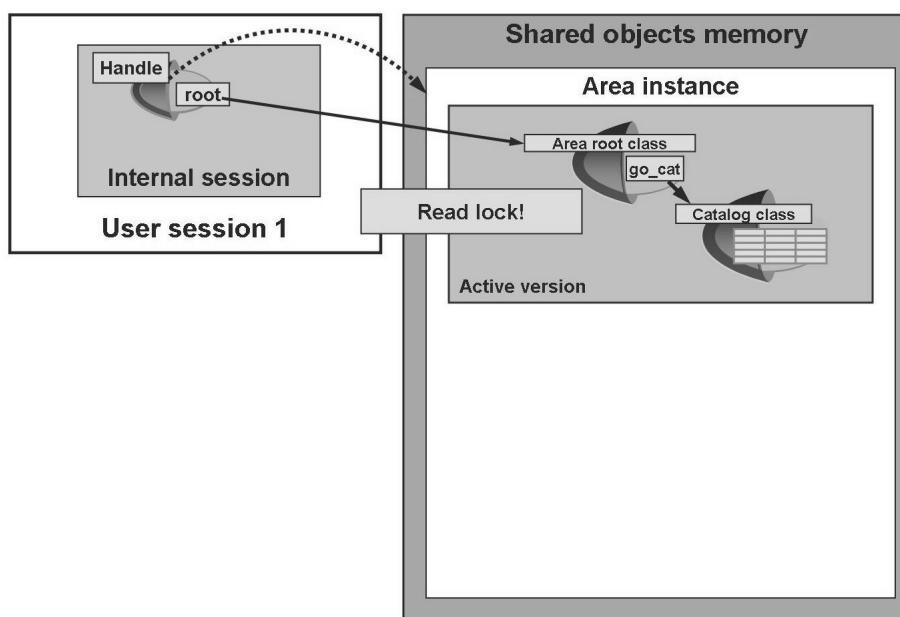
Once the read activity is complete, the application releases the read lock. Method DETACH of the area handle is available for this purpose. The read lock is also released automatically when the internal session is closed.

## States of Area Instance Versions

When you create an area, you can specify that several versions of an area instance are allowed. What exactly do these area instance versions mean? To answer this question, we examine an example in the next section.

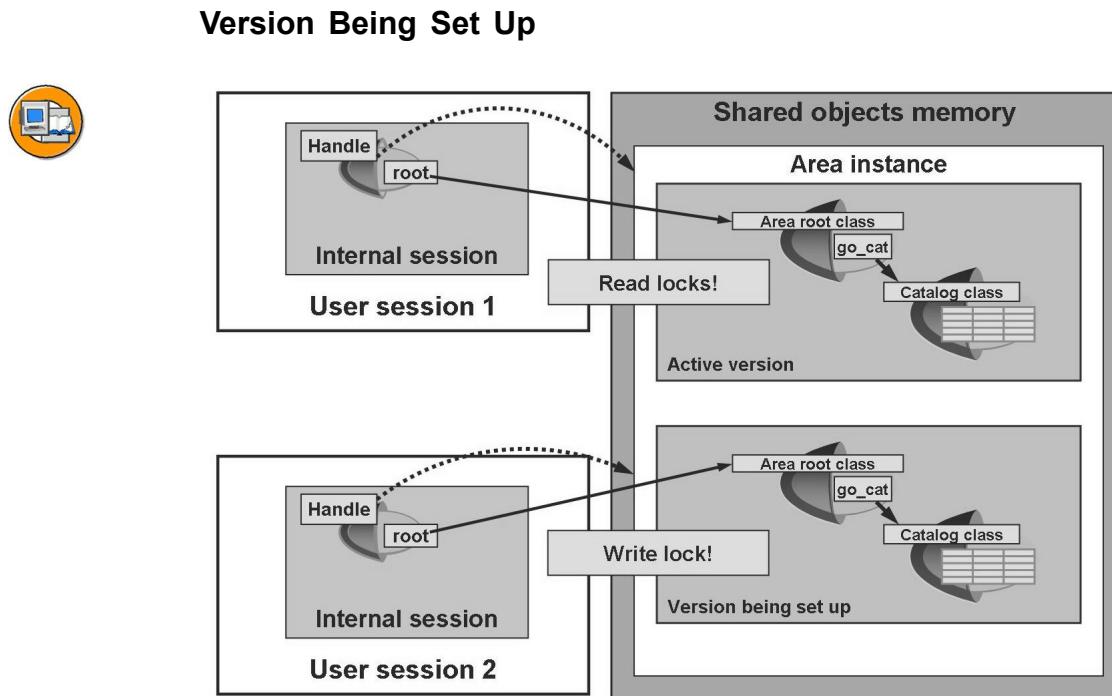


### Active Version



**Figure 76: Setting a Read Lock on the Active Version**

This is the normal state: An area instance has been set up. Once the setup is complete (using method `DETACH_COMMIT`) and the system sends a database commit, the area instance version is active.

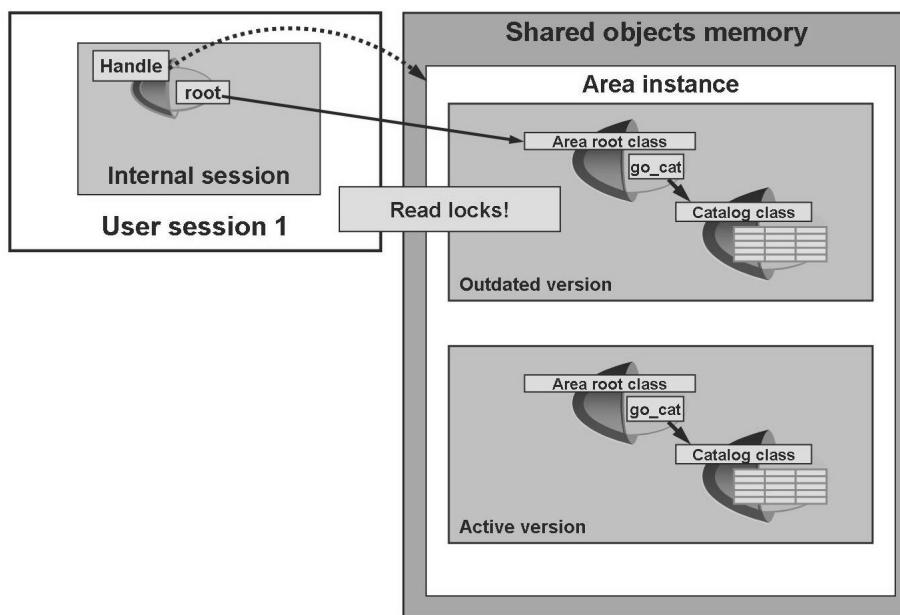


**Figure 77: Version Being Set Up**

If the *Number of Versions* attribute of the area is set appropriately, additional versions of the area instance can exist in addition to the active version.

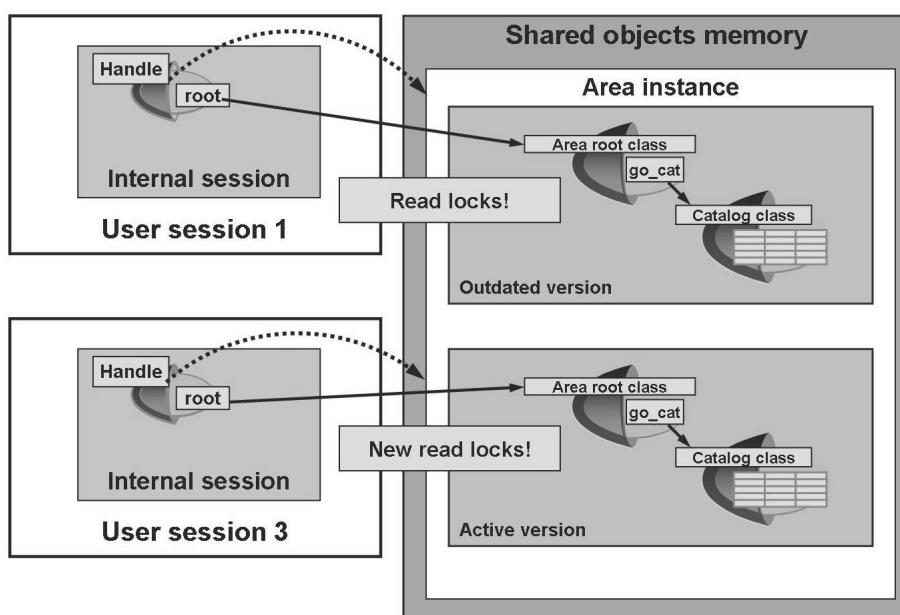
When a new catalog is being set up, several temporary versions of the same area instance exist concurrently. As soon as an application sets a change lock for an area instance, a “version being set up” is created and exists in parallel to the active version.

## Outdated Version



**Figure 78: Write Complete – Previous Version Is Outdated**

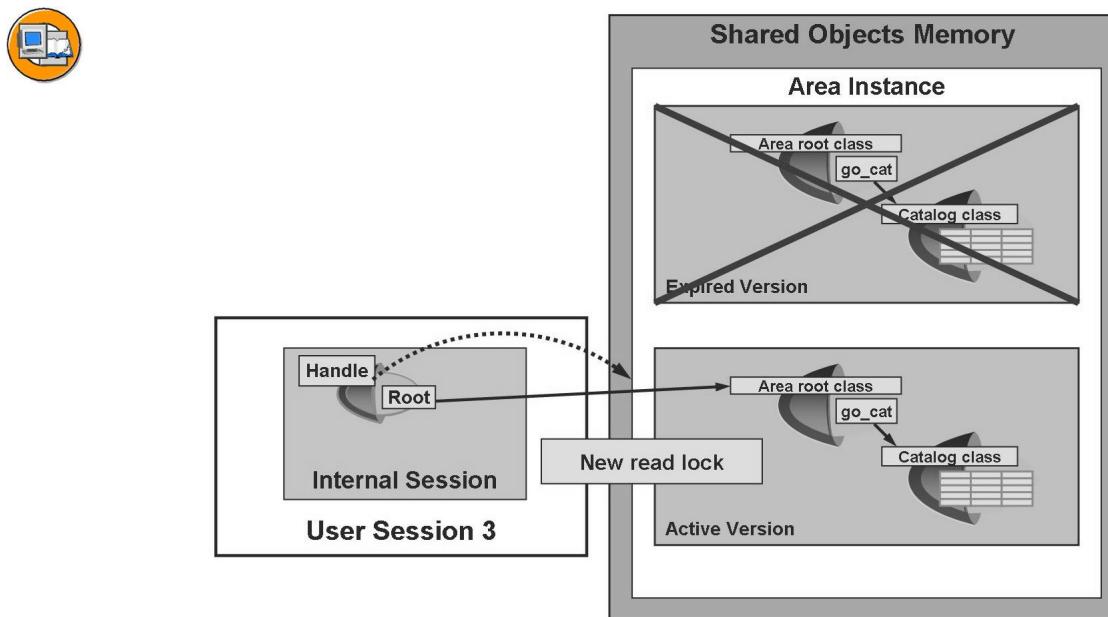
If the setup of the new version is completed during a read access of the currently active version, the version being set up becomes active. The previously active version is given the attribute *Outdated*.



**Figure 79: New Read Locks for the New Active Version**

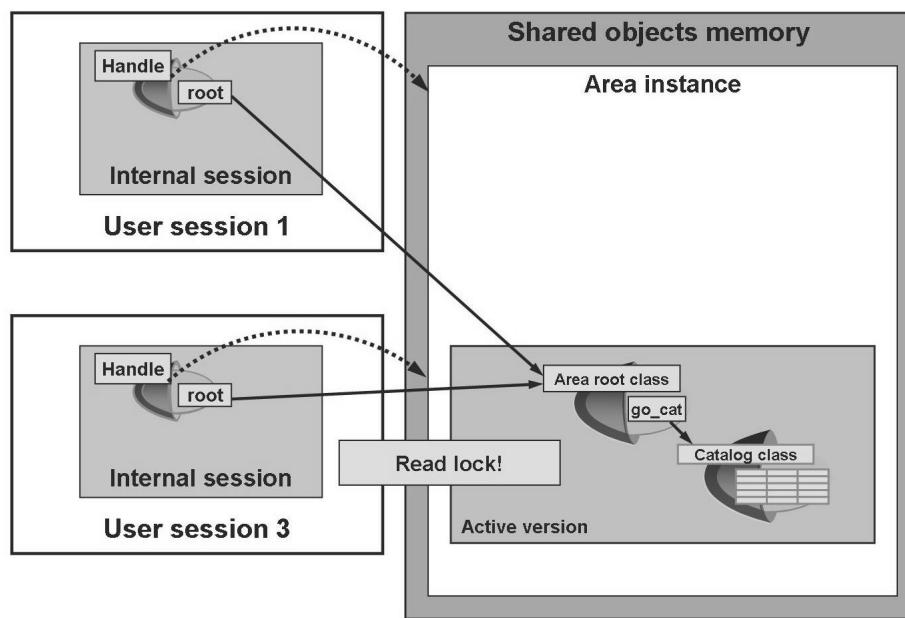
The read locks for the outdated version remain until the read operations are complete. In contrast, new read locks for the area instance are always set for the active version. Therefore, two different readers may get two different read results in this case.

### Expired Version



**Figure 80: No More Read Blocks for Outdated Version: Version Expired**

When the last existing read lock for an outdated version is removed, the version expires. It is then deleted by the garbage collector. You cannot set any locks for expired versions, nor are they used to determine the number of versions (the latter is important if the number of area instance versions has been restricted).



**Figure 81: New Read Locks for Active Version**

New read locks are always set for the active version. There can only be one active version for each area instance. Depending on the maximum number of versions, several outdated versions for which read locks are still set may exist in parallel.

## Exercise 5: Optional: Using Shared Objects

### Exercise Objectives

After completing this exercise, you will be able to:

- Create a shared objects area
- Write data to a shared objects area
- Read data from a shared objects area

### Business Example

You want to run an Internet shop based on an SAP system. To give your Internet customers the fastest possible access to your catalog, you save it in the shared objects memory.

You will need a “write program” to manage and describe the catalog and develop a test application to read the data.

#### Template:

none

#### Solution:

CL\_BC402\_SHS\_ROOT (global class)  
CL\_BC402\_SHS\_CATALOG (global class)  
BC402\_SHS\_WRITE\_CATALOG (executable program)  
BC402\_SHS\_READ\_CATALOG (executable program)

### Task 1:

Create an area in shared objects memory (suggested name: **ZCL\_BC402\_##\_AREA**, where ## is your group number).

1. Use transaction SHMA to create an area in shared memory. This area should have the following attributes:

Client-specific Area
With Versioning
Displacement not Possible

The system automatically prompts you to specify a root class. Create an area root class (suggested name: **ZCL\_BC402\_##\_ROOT**).

*Continued on next page*

2. Implement the area root class. It should merely contain a reference to the catalog class that you will create in the next exercise.

### Task 2:

Create a global class where the catalog data is to be saved (suggested name: **ZCL\_BC402\_##\_CATALOG**). This class requires a method for writing the catalog data and another method for retrieving the data. It also requires a global attribute: an internal table that saves the catalog data.

1. Create the global class for the catalog data. Note that this class has to be shared memory-enabled.
2. Create a private instance attribute in the class (suggested name: **MT\_CATALOG**). Type it with a table type that has structure SDYN\_CONN as a line type.
3. Create a public instance method to fill the catalog (suggested name: **FILL\_CATALOG**). This method requires an importing parameter that will be given the internal table with data.  
Implement the method.
4. Create a second public instance method that can be used to answer flight date queries (suggested name: **GET\_FLIGHTS**).

This method requires one importing parameter each for the departure and destination cities, as well as two additional parameters for the earliest and latest possible flight dates. The method also needs an exporting parameter that returns the matching flights.

Implement the method.

### Task 3:

Create an executable program that builds the area in shared objects memory (suggested name: **ZBC402\_##\_WRITE\_CATALOG**). This program is also supposed to read the catalog data from the database and pass it on to the appropriate class.

1. Create an executable program.
2. Define three reference variables: one each for the area handle, the area root class, and the catalog class.
3. Create an area, an instance of the catalog class, and an instance of the area root class. Set up the area.

Use the Shared Objects Monitor (Transaction SHMM) to trace the setup of the area and the set blocks.

Read the necessary catalog data from tables SPFLI and SFLIGHT.

*Continued on next page*

4. Don't forget to delete the write lock after the data is written successfully.

### Task 4:

Implement a read program to read data from the area you just created (suggested name: **ZBC402\_##\_READ\_CATALOG**). Output the data in a simple user dialog (using the provided function module BC402\_DISPLAY\_TABLE, for example).

1. Create an executable program.
2. Define a selection screen in which the user can enter the departure and destination cities. The user should also be able to enter an earliest and latest possible flight date.
3. Make sure the data is read from the catalog object. Output the data in the user dialog.

## Solution 5: Optional: Using Shared Objects

### Task 1:

Create an area in shared objects memory (suggested name: **ZCL\_BC402\_##\_AREA**, where ## is your group number).

1. Use transaction SHMA to create an area in shared memory. This area should have the following attributes:

Client-specific Area
With Versioning
Displacement not Possible

The system automatically prompts you to specify a root class. Create an area root class (suggested name: **ZCL\_BC402\_##\_ROOT**).

2. Implement the area root class. It should merely contain a reference to the catalog class that you will create in the next exercise.
  - a) See the source code excerpt from the model solution.

### Task 2:

Create a global class where the catalog data is to be saved (suggested name: **ZCL\_BC402\_##\_CATALOG**). This class requires a method for writing the catalog data and another method for retrieving the data. It also requires a global attribute: an internal table that saves the catalog data.

1. Create the global class for the catalog data. Note that this class has to be shared memory-enabled.
  - a) Create the class in the *Class Builder*. Use the sample solution as a guide.
2. Create a private instance attribute in the class (suggested name: **MT\_CATALOG**). Type it with a table type that has structure SDYN\_CONN as a line type.
  - a) See the source code excerpt from the model solution.
3. Create a public instance method to fill the catalog (suggested name: **FILL\_CATALOG**). This method requires an importing parameter that will be given the internal table with data.

*Continued on next page*

Implement the method.

- a) See the source code excerpt from the model solution.
4. Create a second public instance method that can be used to answer flight date queries (suggested name: **GET\_FLIGHTS**).

This method requires one importing parameter each for the departure and destination cities, as well as two additional parameters for the earliest and latest possible flight dates. The method also needs an exporting parameter that returns the matching flights.

Implement the method.

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

### Task 3:

Create an executable program that builds the area in shared objects memory (suggested name: **ZBC402\_##\_WRITE\_CATALOG**). This program is also supposed to read the catalog data from the database and pass it on to the appropriate class.

1. Create an executable program.
  - a) Carry out this step in the usual manner.
2. Define three reference variables: one each for the area handle, the area root class, and the catalog class.
  - a) See the source code excerpt from the model solution.
3. Create an area, an instance of the catalog class, and an instance of the area root class. Set up the area.

Use the Shared Objects Monitor (Transaction SHMM) to trace the setup of the area and the set blocks.

Read the necessary catalog data from tables SPFLI and SFLIGHT.

- a) See the source code excerpt from the model solution.
4. Don't forget to delete the write lock after the data is written successfully.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

## Task 4:

Implement a read program to read data from the area you just created (suggested name: **ZBC402\_##\_READ\_CATALOG**). Output the data in a simple user dialog (using the provided function module BC402\_DISPLAY\_TABLE, for example).

1. Create an executable program.
  - a) Carry out this step in the usual manner.
2. Define a selection screen in which the user can enter the departure and destination cities. The user should also be able to enter an earliest and latest possible flight date.
  - a) See the source code excerpt from the model solution.
3. Make sure the data is read from the catalog object. Output the data in the user dialog.
  - a) See the source code excerpt from the model solution.

## Result

### Class CL\_BC402\_SHS\_CATALOG: Public Section

```
class CL_BC402_SHS_CATALOG definition
public
final
create public
shared memory enabled .

*** public components of class CL_BC402_SHS_CATALOG
*** do not include other source files here!!!
public section.

methods FILL_CATALOG
    importing
        !IT_CATALOG type BC402_T_SDYNCONN .
methods GET_FLIGHTS
    importing
        !IV_FROM_CITY type S_FROM_CIT
        !IV_TO_CITY   type S_TO_CITY
        !IV_EARLIEST   type S_DATE
        !IV_LATEST     type S_DATE
    exporting
        !ET_FLIGHTS type BC402_T_SDYNCONN .
```

*Continued on next page*

## Class CL\_BC402\_SHS\_CATALOG: Implementation

```

METHOD fill_catalog.

    gt_catalog = it_catalog.

ENDMETHOD.

METHOD get_flights.

FIELD-SYMBOLS:
    <fs> TYPE sdyn_conn.

LOOP AT gt_catalog ASSIGNING <fs>
    WHERE cityfrom = iv_from_city AND
          cityto   = iv_to_city   AND
          fldate   > iv_earliest AND
          fldate   < iv_latest.

    APPEND <fs> TO et_flights.

ENDLOOP.

ENDMETHOD.

```

## Program BC402\_SHS\_WRITE\_CATALOG

```

REPORT bc402_shs_write_catalog.

DATA:
    go_handle  TYPE REF TO cl_bc402_shs_area,
    go_root    TYPE REF TO cl_bc402_shs_root,
    go_catalog TYPE REF TO cl_bc402_shs_catalog,

    gt_flights TYPE      bc402_t_sdynconn.

START-OF-SELECTION.

go_handle = cl_bc402_shs_area->attach_for_write( ).
```

*Continued on next page*

```

CREATE OBJECT go_root      AREA HANDLE go_handle.
CREATE OBJECT go_catalog AREA HANDLE go_handle.

go_root->mo_catalog = go_catalog.
go_handle->set_root( root = go_root ).

SELECT *
  FROM spfli JOIN sflight
  ON   spfli~carrid = sflight~carrid AND
       spfli~connid = sflight~connid
  INTO CORRESPONDING FIELDS OF TABLE gt_flights.

go_handle->root->mo_catalog->fill_catalog(
  it_catalog = gt_flights ).

go_handle->detach_commit( ).
```

## Program BC402\_SHS\_READ\_CATALOG

```

REPORT bc402_shs_read_catalog.

DATA:
  go_handle      TYPE REF TO    cl_bc402_shs_area,
  gt_flights     TYPE          bc402_t_sdynconn,
  gs_flight      LIKE LINE OF gt_flights,

  gv_startdate  TYPE sydatum,
  gv_enddate    TYPE sydatum.

PARAMETERS:
  pa_from        TYPE spfli-cityfrom,
  pa_to          TYPE spfli-cityto.

SELECT-OPTIONS:
  so_date        FOR sy-datum NO-EXTENSION.

AT SELECTION-SCREEN.

READ TABLE so_date INDEX 1.
IF so_date IS INITIAL.
  so_date-low = sy-datum.
  so_date-high = sy-datum + 365.
```

*Continued on next page*

```
ENDIF.  
gv_startdate = sy-datum.  
gv_enddate   = sy-datum + 365.  
  
IF so_date-low > sy-datum.  
  gv_startdate = sy-datum.  
ENDIF.  
IF so_date-high < gv_enddate.  
  gv_enddate = so_date-high.  
ENDIF.  
  
START-OF-SELECTION.  
  
go_handle = cl_bc402_shs_area->attach_for_read( ).  
  
go_handle->root->mo_catalog->get_flights(  
  EXPORTING  
    iv_from_city = pa_from  
    iv_to_city   = pa_to  
    iv_earliest  = gv_startdate  
    iv_latest    = gv_enddate  
  IMPORTING  
    et_flights   = gt_flights ).  
  
go_handle->detach( ).  
  
CALL FUNCTION 'BC402_DISPLAY_TABLE'  
  CHANGING  
    ct_table = gt_flights.
```



## Lesson Summary

You should now be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program



## Unit Summary

You should now be able to:

- Explain how programs are called from within other programs
- Describe the various options for passing data on to the called program during program calls.
- Describe how the ABAP runtime environment executes programs
- Describe the importance of the runtime object
- Explain the terms "roll area" and "PXA"
- Explain memory management in the ABAP runtime environment
- Describe how dynamic data objects are managed
- Explain how "Boxed Components" are managed and when they should be used
- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program



Internal Use SAP Partner Only

# Unit 3

## Processing Internal Data

### Unit Overview

In this unit, we learn about specific aspects of processing data in an ABAP program.

You learn how arithmetic expressions are interpreted by the ABAP runtime environment, as well as the critical impact the types of the involved data objects can have on the result. You learn about techniques for processing byte strings and character strings - especially string functions, string templates, and regular expressions. Then we find out what "Unicode" means, and learn how the requirement for Unicode capability affects the ABAP syntax.

You learn about data references and field symbols, two special data objects that you can use as pointer variables – particularly in dynamic programming, but also when using internal tables.

In the two lessons on internal tables, you first learn how to use the sorted and hashed table types correctly. This is followed by special techniques for using internal tables, particularly the use of field symbols and data references to access table content directly.

### Unit Objectives



After completing this unit, you will be able to:

- Explain the differences between integer, floating-point, and fixed-point arithmetic
- Explain how the runtime environment evaluates arithmetic expressions
- Understand Unicode and the stricter syntax requirements it imposes
- Use operators and functions to analyze byte strings and character strings
- Use string templates to define character strings
- Use regular expressions to define search patterns
- Describe the differences between standard, sorted, and hashed table types
- Tell the difference between key accesses and index accesses to internal tables
- Use the different table types correctly
- Use the BINARY SEARCH addition correctly

- Use the DELETE ADJACENT DUPLICATES and COLLECT statements correctly
- Describe the function of secondary keys for internal tables and apply them correctly.
- Explain the syntax for working with data references and field symbols
- Use field symbols and data references to access the content of internal tables

## Unit Contents

Lesson: Arithmetic Expressions .....	127
Exercise 6: Optional: Using Numeric Data Types.....	135
Lesson: Processing Byte Strings and Character Strings .....	146
Exercise 7: Processing Character Strings .....	169
Exercise 8: Splitting Character Strings in Internal Tables .....	181
Lesson: Table Types and Their Use.....	189
Exercise 9: Table Types .....	205
Lesson: Special Techniques for Using Internal Tables.....	213
Exercise 10: Using Special Techniques for Internal Tables .....	225
Lesson: Using Data References and Field Symbols .....	235
Exercise 11: Field Symbols and Data References with Internal Tables .....	243

# Lesson: Arithmetic Expressions

## Lesson Overview

In this lesson, you learn how ABAP evaluates arithmetic expressions. You learn how the result of a calculation can depend on which arithmetic operation is used, as well as how the numeric types of the involved data objects influence the selection of the arithmetic operations.

This information helps you to avoid frequent errors when formulating arithmetic expressions.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain the differences between integer, floating-point, and fixed-point arithmetic
- Explain how the runtime environment evaluates arithmetic expressions

## Business Example

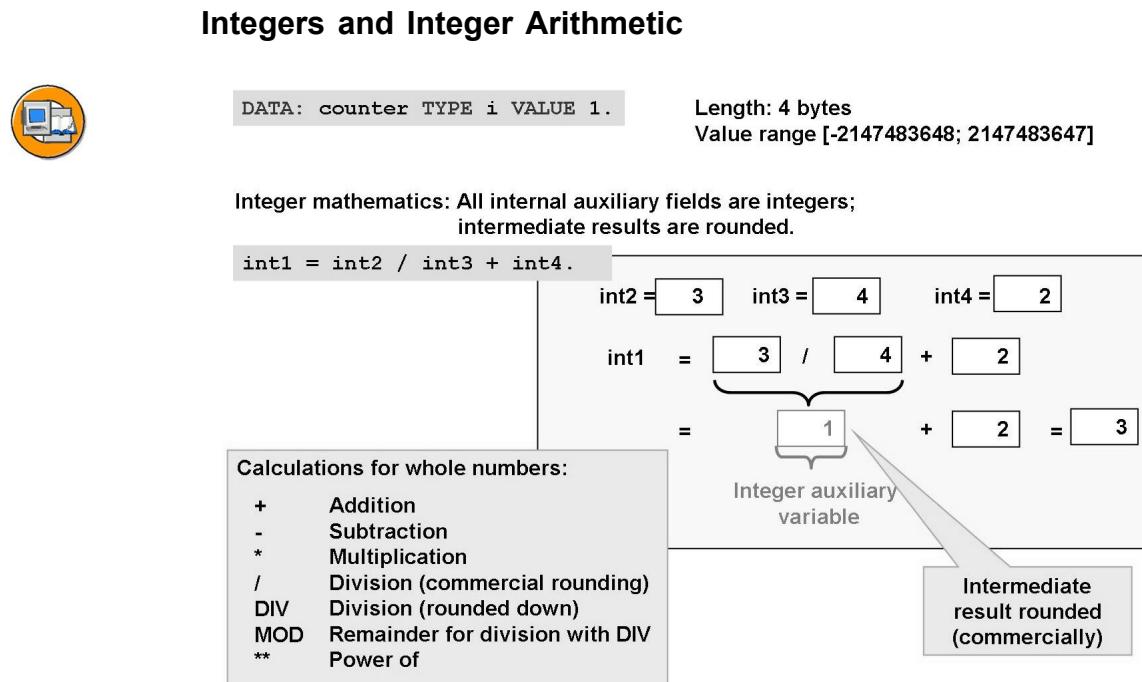
You want to avoid unexpected inaccuracies in calculation results. Therefore, you want to find out in detail how arithmetic expressions are evaluated and processed in ABAP.

## Numeric Data Objects and Arithmetic Expressions

There are three numeric data types in ABAP, which differ in their inner representation of values and their maximum value range. Therefore, which data type you choose for a data object depends on its required value range. In addition, each of the three data types is linked with its own **arithmetic**, which is used for calculations involving data objects of the respective type. We differentiate between the following:

- Type i – integer arithmetic
- Type p – fixed point arithmetic
- Type f – floating point arithmetic

We examine the three numeric data types and their corresponding arithmetic methods in more detail below. We initially assume that “pure” arithmetic expressions are involved – that is, expressions in which all the involved data objects have the same type. We then examine mixed arithmetic expressions and learn the how the runtime environment determines which arithmetic to use.



**Figure 82: Integers and Integer Arithmetic**

Whole numbers with type i are represented internally as binary numbers with a length of 4 bytes. They therefore have a value range of  $-2^{31}+1$  to  $2^{31}-1$ .

Calculations with integer arithmetic are faster than calculations with fixed point or floating point arithmetic, because they are executed directly using the processor's arithmetic command set.

Non-integer results are rounded to the next whole number.



**Caution:** In contrast to many other programming languages, integer arithmetic in ABAP **rounds to the next whole number**.

Example:

```
DATA int TYPE i.

int = 4 / 10.      " ==> 0
int = 5 / 10.      " ==> 1
```

Moreover – again in contrast to many other programming languages – all **interim results are also rounded**.

Example:

```
DATA int TYPE i.
```

```
int = 10 * ( 4 / 10 ).      " ==> 0
int = ( 10 * 4 ) / 10 .     " ==> 4
```

If an interim result lies outside the value range of data type i, a runtime error occurs.

## Floating Point Numbers and Binary Floating Point Arithmetic

Floating point numbers are represented by binary precision floating point numbers. Floating point numbers are normalized, and both the exponent and the mantissa are stored in binary form. This representation complies with the IEEE norm for double precision floating point numbers (IEEE-754).

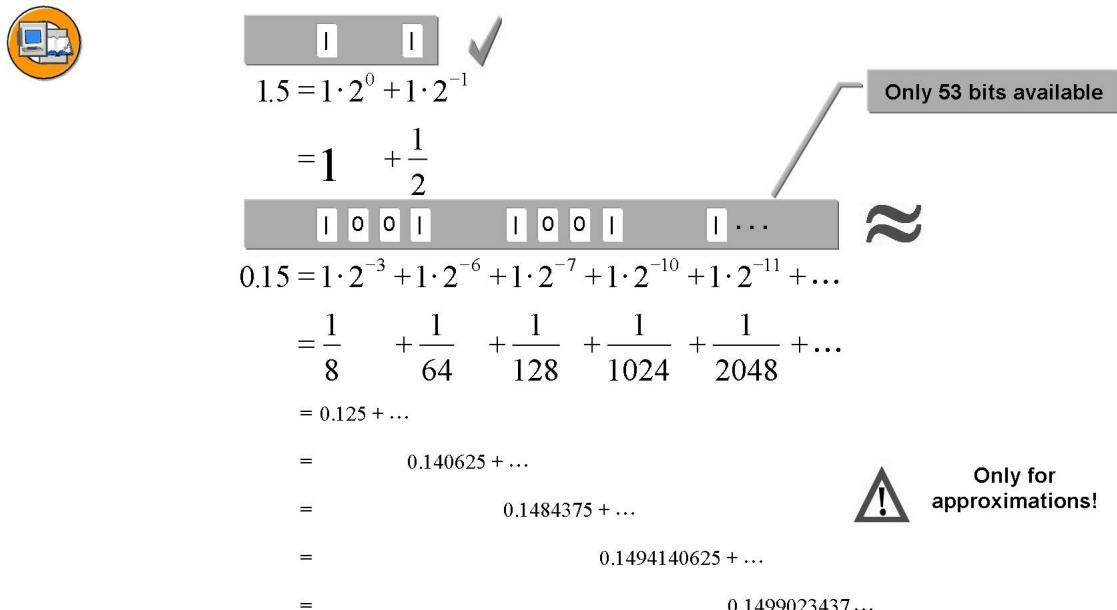


Figure 83: Binary Floating Point Arithmetic



**Caution:** The binary floating point operations of the relevant processors are used for calculations in floating point arithmetic. Since algorithms are converted to binary, **inaccuracies** can occur. The extent and effect of these inaccuracies are difficult to estimate.

You can demonstrate this in the following example:

Suppose you want to calculate 7.27% of 73050 and display the result accurate to two decimal places. The answer should be 5310.74 ( $73050 * 0.0727 = 5310.735$ ). However, the program returns the following:

```
DATA: float TYPE f,
```

```

    pack TYPE p DECIMALS 2.
float = 73050 * '0.0727'.   " ==> 5.310734999999997E+03
pack = float.
WRITE pack.                  " ==> 5310.73

```

Due to these effects, floating point arithmetic is unsuitable for business calculations, because these calculations have to be exact "to the penny" and comply with legal rounding regulations. You should, therefore, use floating-point numbers for approximations only. When you compare floating point numbers, always use intervals, and always round at the end of your calculations.

The main advantage of floating point numbers is their large range of values, from 2.2250738585072014E-308 to 1.7976931348623157E+308, including both positive and negative numbers and zero. In addition, you must use floating point numbers for special aggregation functions of the SELECT statement.

## Packed Numbers and Fixed Point Arithmetic

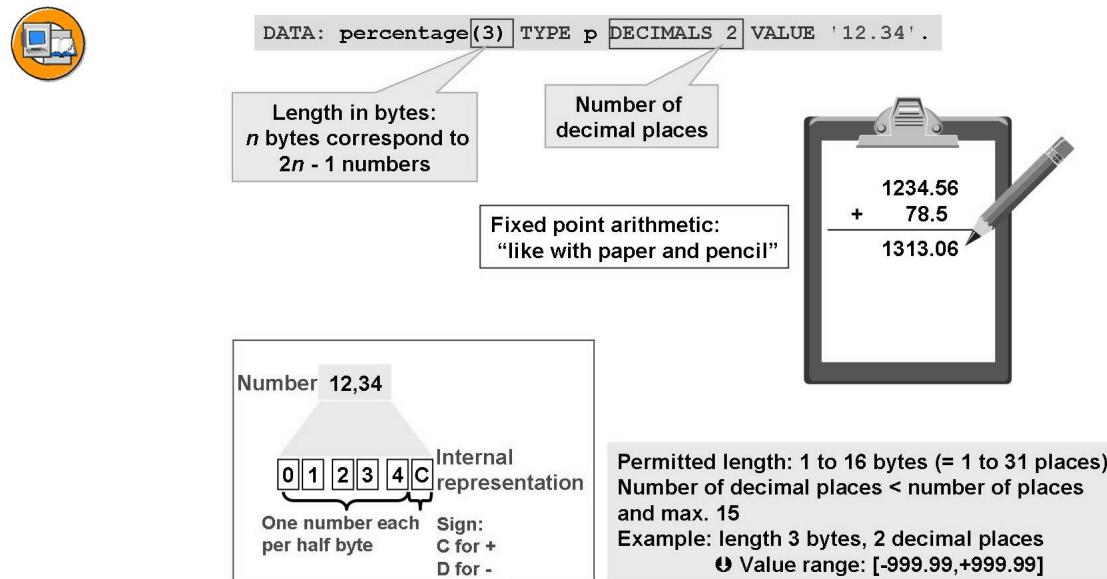


Figure 84: Packed Numbers and Fixed Point Arithmetic

**Packed numbers** were introduced in ABAP to expand the value range to decimal numbers, yet avoid the rounding problems associated with floating point numbers.

When you define packed numbers, you can determine the length of the number, which you specify in bytes. The connection between the value range and the length is determined from the internal representation of packed numbers: Each decimal digit is represented by a half-byte; the last byte is reserved for the plus or minus sign. Accordingly, the number of decimal digits is derived from the number of bytes according to the following rule:  $n_{\text{digits}} = 2 n_{\text{bytes}} - 1$ . The DECIMALS addition

determines how many of these decimal digits are handled as decimal places. The maximum number of decimal places is 15, however. Moreover, the number must contain at least one digit to the left of the decimal point.

Decimal point-aligned **fixed point arithmetic** is used for calculations. Packed numbers are thus well-suited to business calculations where the correct rounding for the specified number of decimal places is important. The algorithm for this arithmetic is similar to using "pencil and paper". Interim results initially use packed numbers with 31 decimal places. Should an overflow occur, the entire expression is calculated again with an internal accuracy of 63 decimal places. If another overflow occurs, an exception is raised (and can be handled).



**Hint:** You can switch off fixed point arithmetic in the program attributes. If you do, the DECIMALS addition for defining a packed number affects only the output of the number. Internally, all numbers are interpreted as integers, regardless of the position of the decimal point. The fixed point arithmetic option is always selected by default. You should always accept this value and use packed numbers for business calculations.

## Decimal Floating Point Numbers

**Decimal floating point numbers** are introduced in SAP NetWeaver 7.0 EhP2 to overcome the drawbacks of binary floating point numbers: Type f data objects cannot represent every decimal number precisely due to the internal binary representation. Results of calculations may depend on the platform, there is no rounding to a specific number of decimal places, division by powers of 10 is inexact, and no uniform behavior exists across database systems.

Decimal floating point numbers have a range larger than type f, while the calculation accuracy is identical to type p. Two different internal types exist:



Decfloat16

- Memory consumption: 8 bytes
  - Digits: 16
  - Range: 1.0E-383 through 9.99999999999999E+384

Decfloat34



**Hint:** In the ABAP Dictionary, corresponding data types have been defined (DF16\_<A> and DF34\_<A>, where <A> can be DEC, RAW, or SCL).

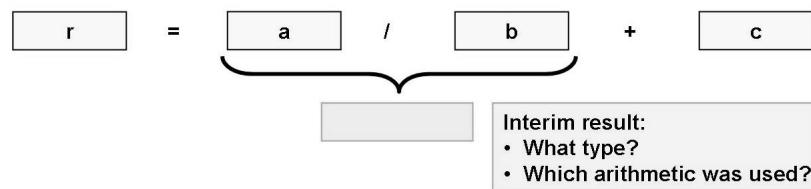
**Decimal floating point arithmetic** is used for calculations. The algorithm for this arithmetic is similar to using "pencil and paper". Interim results use numbers with 16 (decfloat16) or 34 (decfloat34) decimal places. The largest number in the arithmetic expression defines the exponent. Numbers that are too small (according to the exponent and the number of decimals defined by the type) are rounded.

Decfloat16: Should an overflow occur, the entire expression is calculated again with an accuracy and number range based on type decfloat34. The same applies, if the number of decimals (16) is too small to store the intermediate result without having to round. If another overflow occurs, an exception is raised (and can be handled).

### Mixed Arithmetic Expressions

An arithmetic expression can contain a mix of all the numeric data types. If an arithmetic expression contains at least one data object with a numeric type, it can also contain character-type data objects, provided their contents can be interpreted as numbers. The values contained in these character-type objects are converted to numeric type objects.

The arithmetic to use is determined based on the data types before the calculation is performed. Different types of arithmetic have different degrees of internal accuracy. Thus, the same arithmetic expression can lead to different results, depending on which data types are involved. The examples in the following graphic illustrate this.



Conversion in mixed expressions:

Types				Interim result	Values			
r	a	b	c		a	b	c	r
i	i	i	i	i	201	200	0	1
p decimals 2	i	i	p	p	201	200	0	1.01
p decimals 3	i	i	p	p	201	200	0	1.005
p decimals 2	i	i	f	f	201	200	0	1.00
p decimals 3	i	i	f	f	201	200	0	1.005
p decimals 3	i	i	i	p	201	200	0	1.005

Figure 85: Results of Arithmetic for Mixed Expressions

The data types of **all** operands, along with the data type of the result field, are taken into account when determining the arithmetic. Once the arithmetic has been determined, all operands are converted to the numeric type that corresponds to the chosen arithmetic. The system performs the calculation using these converted values and ultimately converts the result to the desired result type.



**Hint:** Bear in mind that conversions affect performance. To avoid them, try to make sure that all operands and the result field have the same numeric type.



Types			Values				Arithmetic
r	a	b	c	a	b	c	r
i	i	i	i	201	200	0	1
p decimals 2	i	i	p	201	200	0	1.01
p decimals 3	i	i	p	201	200	0	1.005
p decimals 2	i	i	f	201	200	0	1.00
p decimals 3	i	i	f	201	200	0	1.005
p decimals 3	i	i	i	201	200	0	1.005

The data types of all the components combined determine the arithmetic of the whole expression

#### Rules:

- Fixed point arithmetic is the default
- Integer arithmetic is used only if **all** the components are integers
- Floating point arithmetic is used whenever **at least one** component is floating point

**Figure 86: Arithmetic Used for Mixed Expressions**

The following rules apply to determining the arithmetic:

- **Decimal floating point arithmetic** is used if **at least** one operand or the result field is of data type decfloat16 or decfloat34.
- **Binary floating point arithmetic** is used if **at least** one operand or the result field is of data type f and none of the operands is of data type decfloat16 or decfloat34.
- **Fixed point arithmetic** is used if **at least** one operand or the result field is of data type p and none of the operands or the result field is of data type f, decfloat16, or decfloat34.
- **Integer arithmetic** is used in all other cases, i.e. if **all** operands and the result field are of data type i.



**Caution:** When trigonometry functions and other scientific functions, logarithms and exponential functions (including the `**` operator) are used, binary floating point arithmetic or decimal floating point arithmetic is used for calculation.

Example:

```
DATA int TYPE i.  
  
int = ( 1 / 4 ) * ( 2 * 2 ).           " ==> 0  
int = ( 1 / 4 ) * ( 2 ** 2 ).          " ==> 1
```



**Caution:** You should also exercise caution when using numeric literals with large amounts. They only have data type i if they do not exceed the value range of data type i.

Example:

```
DATA int TYPE i.  
int = 1000000000 / 300000000 * 3.      " ==> 9  
int = 1000000000 / 300000000 * 3.        " ==> 10
```

## Exercise 6: Optional: Using Numeric Data Types

### Exercise Objectives

After completing this exercise, you will be able to:

- Create data objects with numeric data types
- Perform calculations with numeric data types

### Business Example

As part of a new application you are developing, you have to ensure that the numeric data types are used correctly. To do so, you develop test programs.

Create three programs to test the effects of the following operations on the three numeric data types (one data type for each program): addition (+), subtraction (-), division (/), multiplication (\*), integer division (DIV), modulus (remainder of integer division) (MOD), and powers (\*\*).

#### Template:

BC402\_IDT\_CALC

#### Solution:

BC402\_IDT\_CALC\_I (whole numbers)

BC402\_IDT\_CALC\_P (packed numbers)

BC402\_IDT\_CALC\_F (floating-point numbers)

### Task 1:

Copy program BC402\_IDT\_CALC to the name **ZBC402\_##\_CALC\_I**, where ## is your group number. This program has a selection screen where two values can be entered. The entries are then copied to type i data objects. Use these two values to implement the calculation operations and output the results in a list.

1. Copy the template and all its subcomponents.
2. Perform the calculations in the START-OF-SELECTION event block in sequence. Display the result of each calculation in the list. Generate a list line for each calculation operation and output both values, the respective operator, and the result.

When creating your list, use the following example as a guide:

*Continued on next page*

## List Output

Rechnen mit Integer-Zahlen

Operand 1 =

2

Operand 2 =

1

2 +	1 =	3
2 -	1 =	1
2 *	1 =	2
2 /	1 =	2
2 DIV	1 =	2
2 MOD	1 =	0
2 **	1 =	2

3. Activate and test the program. Does the result match your expectations? What do you notice?

---



---



---



---

## Task 2:

Copy your program ZBC402\_##\_CALC\_I (or use the model solution from the previous exercise, BC402\_IDS\_CALC\_I) and give it the name **ZBC402\_##\_CALC\_F**. Modify the program to test the effects of the calculation operations described above on **floating-point numbers**.

1. Copy the program.
2. Change the typing of data objects gv\_val1, gv\_val2, and gv\_result so the calculations are now carried out with type f data objects.

*Continued on next page*

3. Activate and test your program. What do you observe?

---

---

---

### Task 3:

Copy your program ZBC402\_##\_CALC\_I (or use the model solution from the first exercise, BC402\_IDS\_CALC\_I) and give it the name **ZBC402\_##\_CALC\_P**. Modify the program to test the effects of the calculation operations described above on **packed numbers**.

1. Copy the program.
2. Change the typing of data objects gv\_val1, gv\_val2, and gv\_result so the calculations are now carried out with type p data objects. Choose a suitable number of decimal places (not 0) for each object.
3. Activate and test your program. What do you observe?

---

---

---

4. Remove the *Fixed Point Arithmetic* attribute in the program attributes. Test your program. What do you observe?

---

---

---

## Solution 6: Optional: Using Numeric Data Types

### Task 1:

Copy program BC402\_IDT\_CALC to the name **ZBC402\_##\_CALC\_I**, where ## is your group number. This program has a selection screen where two values can be entered. The entries are then copied to type i data objects. Use these two values to implement the calculation operations and output the results in a list.

1. Copy the template and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Perform the calculations in the START-OF-SELECTION event block in sequence. Display the result of each calculation in the list. Generate a list line for each calculation operation and output both values, the respective operator, and the result.

When creating your list, use the following example as a guide:

### List Output

Rechnen mit Integer-Zahlen

Operand 1 =	2
Operand 2 =	1
<hr/>	
2 +	1 = 3
2 -	1 = 1
2 *	1 = 2
2 /	1 = 2
2 DIV	1 = 2
2 MOD	1 = 0
2 **	1 = 2

- a) See source code excerpt from the model solution.
3. Activate and test the program. Does the result match your expectations? What do you notice?

**Answer:** When dividing with /, commercial rounding is applied to the result.

*Continued on next page*

## Task 2:

Copy your program ZBC402\_##\_CALC\_I (or use the model solution from the previous exercise, BC402\_IDS\_CALC\_I) and give it the name **ZBC402\_##\_CALC\_F**. Modify the program to test the effects of the calculation operations described above on **floating-point numbers**.

1. Copy the program.
  - a) Carry out this step in the usual manner.
2. Change the typing of data objects gv\_val1, gv\_val2, and gv\_result so the calculations are now carried out with type f data objects.
  - a) See the source code.
3. Activate and test your program. What do you observe?

**Answer:** The accuracy depends on the absolute size of the result - 17 places are always displayed.

## Task 3:

Copy your program ZBC402\_##\_CALC\_I (or use the model solution from the first exercise, BC402\_IDS\_CALC\_I) and give it the name **ZBC402\_##\_CALC\_P**. Modify the program to test the effects of the calculation operations described above on **packed numbers**.

1. Copy the program.
  - a) Carry out this step in the usual manner.
2. Change the typing of data objects gv\_val1, gv\_val2, and gv\_result so the calculations are now carried out with type p data objects. Choose a suitable number of decimal places (not 0) for each object.
  - a) See the source code excerpt from the model solution.
3. Activate and test your program. What do you observe?

**Answer:** Commercial rounding is now performed for the last place of the result field. You can define the accuracy of the calculation by typing the result field.

4. Remove the *Fixed Point Arithmetic* attribute in the program attributes. Test your program. What do you observe?

**Answer:** The results are not correct, because the position of the decimal point was only considered in the input and output, but not during calculation.

## Result

Source code excerpt from the model solution:

*Continued on next page*

## Calculating with Integers

```

REPORT  bc402_ids_calc_i.

DATA:
  gv_result    TYPE i,                      " result variable
  gv_val1      TYPE i,                      " first operand
  gv_val2      TYPE i.                     " second operand

PARAMETERS:
  pa_val1      TYPE i DEFAULT 5,           " input variable 1
  pa_val2      TYPE i DEFAULT 3.           " input variable 2

START-OF-SELECTION.

* copy from parameters to data objects
  gv_val1 = pa_val1.
  gv_val2 = pa_val2.

  WRITE: / text-001.  " <-- 'calculate with integer data'

  SKIP 1.
  WRITE: / text-002, gv_val1, " <-- 'operand 1 = '
        / text-003, gv_val2. " <-- 'operand 2 = '

  SKIP 2.

* addition:      a + b
  gv_result = gv_val1 + gv_val2.
  WRITE: / gv_val1, ' + ', gv_val2, ' = ', gv_result.
  SKIP 1.

* subtraction:   a - b
  gv_result = gv_val1 - gv_val2.
  WRITE: / gv_val1, ' - ', gv_val2, ' = ', gv_result.
  SKIP 1.

* division:      a / b
  gv_result = gv_val1 / gv_val2.
  WRITE: / gv_val1, ' / ', gv_val2, ' = ', gv_result.
  SKIP 1.

* multiplication: a * b
  gv_result = gv_val1 * gv_val2.

```

*Continued on next page*

```

        WRITE: / gv_val1, ' * ', gv_val2, ' = ', gv_result.
        SKIP 1.

        * div
        gv_result = gv_val1 DIV gv_val2.
        WRITE: / gv_val1, 'div', gv_val2, ' = ', gv_result.
        SKIP 1.

        * mod
        gv_result = gv_val1 MOD gv_val2.
        WRITE: / gv_val1, 'mod', gv_val2, ' = ', gv_result.
        SKIP 1.

        * power
        gv_result = gv_val1 ** gv_val2.
        WRITE: / gv_val1, '** ', gv_val2, ' = ', gv_result.
        SKIP 1.
    
```

## Calculating with Floating Point numbers

```

REPORT bc402_ids_calc_f.

DATA:
  gv_result  TYPE f,                      " result variable
  gv_val1    TYPE f,                      " first operand
  gv_val2    TYPE f.                      " second operand

PARAMETERS:
  pa_val1    TYPE i DEFAULT 5,           " input variable 1
  pa_val2    TYPE i DEFAULT 3.           " input variable 2

START-OF-SELECTION.

* copy from parameters to data objects
  gv_val1 = pa_val1.
  gv_val2 = pa_val2.

  WRITE: / text-001.    " <-- 'calculate with float data'

  SKIP 1.
  WRITE: / text-002, gv_val1, " <-- 'operand 1 = '

```

*Continued on next page*

```

/ text-003, gv_val2. " <-- 'operand 2 = '

SKIP 2.

* addition:      a + b
gv_result = gv_val1 + gv_val2.
WRITE: / gv_val1, ' + ', gv_val2, ' = ', gv_result.
SKIP 1.

* subtraction:   a - b
gv_result = gv_val1 - gv_val2.
WRITE: / gv_val1, ' - ', gv_val2, ' = ', gv_result.
SKIP 1.

* division:      a / b
gv_result = gv_val1 / gv_val2.
WRITE: / gv_val1, ' / ', gv_val2, ' = ', gv_result.
SKIP 1.

* multiplication: a * b
gv_result = gv_val1 * gv_val2.
WRITE: / gv_val1, ' * ', gv_val2, ' = ', gv_result.
SKIP 1.

* div
gv_result = gv_val1 DIV gv_val2.
WRITE: / gv_val1, 'div', gv_val2, ' = ', gv_result.
SKIP 1.

* mod
gv_result = gv_val1 MOD gv_val2.
WRITE: / gv_val1, 'mod', gv_val2, ' = ', gv_result.
SKIP 1.

* power
gv_result = gv_val1 ** gv_val2.
WRITE: / gv_val1, '** ', gv_val2, ' = ', gv_result.
SKIP 1.

```

## Calculating with Packed Numbers

REPORT bc402\_ids\_calc\_p.

*Continued on next page*

```

DATA:
  gv_result  TYPE p LENGTH 8 DECIMALS 4, " result variable
  gv_val1    TYPE p LENGTH 8 DECIMALS 2, " first operand
  gv_val2    TYPE p LENGTH 8 DECIMALS 3. " second operand

PARAMETERS:
  pa_val1     TYPE i DEFAULT 5,           " input variable 1
  pa_val2     TYPE i DEFAULT 3.          " input variable 2

START-OF-SELECTION.

* copy from parameters to data objects
  gv_val1 = pa_val1.
  gv_val2 = pa_val2.

  WRITE: / text-001.    " <-- 'calculate with packed data'

  SKIP 1.
  WRITE: / text-002, gv_val1, " <-- 'operand 1 = '
        / text-003, gv_val2. " <-- 'operand 2 = '

  SKIP 2.

* addition:      a + b
  gv_result = gv_val1 + gv_val2.
  WRITE: / gv_val1, ' + ', gv_val2, ' = ', gv_result.
  SKIP 1.

* subtraction:   a - b
  gv_result = gv_val1 - gv_val2.
  WRITE: / gv_val1, ' - ', gv_val2, ' = ', gv_result.
  SKIP 1.

* division:      a / b
  gv_result = gv_val1 / gv_val2.
  WRITE: / gv_val1, ' / ', gv_val2, ' = ', gv_result.
  SKIP 1.

* multiplication: a * b
  gv_result = gv_val1 * gv_val2.
  WRITE: / gv_val1, ' * ', gv_val2, ' = ', gv_result.
  SKIP 1.

```

*Continued on next page*

```
* div
gv_result = gv_val1 DIV gv_val2.
WRITE: / gv_val1, 'div', gv_val2, ' = ', gv_result.
SKIP 1.

* mod
gv_result = gv_val1 MOD gv_val2.
WRITE: / gv_val1, 'mod', gv_val2, ' = ', gv_result.
SKIP 1.

* power
gv_result = gv_val1 ** gv_val2.
WRITE: / gv_val1, '** ', gv_val2, ' = ', gv_result.
SKIP 1.
```



## Lesson Summary

You should now be able to:

- Explain the differences between integer, floating-point, and fixed-point arithmetic
- Explain how the runtime environment evaluates arithmetic expressions

# Lesson: Processing Byte Strings and Character Strings

## Lesson Overview

In this lesson, you learn important techniques for processing character-type and byte-type data objects. You also learn how demands for Unicode capability affect the syntax of these statements.



## Lesson Objectives

After completing this lesson, you will be able to:

- Understand Unicode and the stricter syntax requirements it imposes
- Use operators and functions to analyze byte strings and character strings
- Use string templates to define character strings
- Use regular expressions to define search patterns

## Business Example

You want to implement an application that processes and manipulates character strings. Familiarize yourself with the ABAP statements and techniques that enable you to do so.

## Using Character-Type and Byte-Type Data Objects

The definition of which data objects are character-type objects depends on how the characters are represented internally. Accordingly, we first discuss the representation of characters before we turn to character-type data objects and their processing.

## Unicode and the Internal Representation of Characters



Character	ASCII (8 bit)
...	...
a	97
b	98
...	...

} 256

### Problem cases:

- Users from different cultural groups in one system
- Data interchange between systems for different cultural groups

Character	Unicode (16 bit)
...	U+...
a	U+0061
...	U+...
α	U+03B1
...	U+...
會	U+3479
...	U+...

} 65536

### Attributes:

- Complete
- Uniform worldwide

**Figure 87: Unicode Character Set Table**

Characters are represented internally as bit sequences. Up to Release 4.6C, each character in an SAP system corresponded to one byte, which meant 256 characters could be represented simultaneously. Different **code pages** are used for the assignment between bit sequence and character in different cultural groups.

In *SAP Web AS 6.10* and later, the **Unicode** standard is supported, which makes it possible to represent each character internally by 2 bytes in appropriately configured systems. The assignment between character and bit sequence is defined by a Unicode character set (currently ISO/IEC 10646) in this case. Since 2 bytes per character theoretically make 65536 unique characters possible, it is no longer necessary to switch between code pages.

The changed representation of characters also affects the processing of strings. Some operations that were possible in non-Unicode systems lead to incorrect (or ambiguous) results in Unicode systems. As a result, the **syntax check** in the *ABAP Workbench* has been enhanced in *SAP NetWeaver Application Server 6.10* and later with rules that prevent non-Unicode-compatible operations.

The Unicode check can be activated or deactivated individually for each program (*Unicode Check Active* checkbox in the program attributes). If need to set this checkbox to run that program in a Unicode system. If the Unicode check is active in a program, you can still run it in non-Unicode systems.



**Hint:** Note that the *Unicode Check Active* checkbox merely activates an additional check. It has no influence on the conversion of ABAP statements or the available syntax.

The checkbox is set by default for new programs. We recommend activating the Unicode check even if your system does not use Unicode (yet). This not only saves you from having to convert your programs later, but also help you avoid unsafe programming techniques and obsolete statements.

You can use transaction UCHECK to find programs for which the Unicode flag has not been set. In addition, you can also set this flag for several programs at the same time and run the check.

## Byte-Type and Character-Type Data Objects

All elementary data objects with a byte-type data type (x, xstring) are **byte-type data objects**.

All elementary data objects with a character-type data type (c, n, d, t, string) are **character-type data objects**.

In **non-Unicode systems**, byte-type data objects, as well as all flat structures, can be handled as character-type data objects. In **Unicode systems**, in contrast, you have to differentiate clearly between byte-type and character-type data objects. Moreover, you can only handle flat structures as character-type data objects if they consist entirely of character-type components.

## Statements for Character and Byte Processing

The following graphic contains an overview of the ABAP statements for processing character-type data objects:



CONCATENATE	$A B + A P \rightarrow A B A P$	Combine multiple strings
FIND	$A B A P$	Find pattern (first / all occurrences)
REPLACE	$A B A P \rightarrow X B X P$	Replace pattern (first / all occurrences)
SHIFT	$A B A P \rightarrow A P$	Shift left / right
SPLIT	$A B A P \rightarrow A + A P$	Split string in segments
CONDENSE	$A      P \rightarrow A P$	Remove space characters
OVERLAY	$\begin{array}{l} A B P \\ A A A \end{array} \rightarrow A B A P$	Replace characters in string by characters from second string 2 at same position
TRANSLATE	$A B A P \rightarrow a b a p$	Replace characters according to rule
WRITE..TO	$07 3 5 00 \rightarrow 07:35:00$	Format string

Figure 88: Overview - Statements for Character String Processing

In each of these statements, the operands are treated like type c fields, regardless of their actual field type. They are not converted. All of the statements aside from TRANSLATE and CONDENSE set system field sy-subrc.

→ **Note:** As of SAP Net Weaver 7.0, the statement FIND replaces the statement SEARCH.



```
DATA: c_field(4) TYPE c VALUE 'HUGO',
      x_field(4) TYPE x VALUE 'E391B9A2'.

SHIFT c_field BY 2 PLACES [IN CHARACTER MODE].
SHIFT x_field BY 2 PLACES IN BYTE MODE.
```

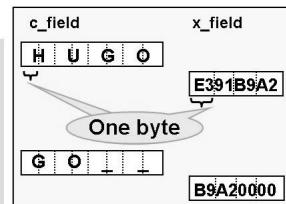


Figure 89: String/Byte Sequence Processing

With the exception of CONDENSE and OVERLAY, you can also use these statements to process **byte-type data objects**. Under Unicode, you have to use the **IN BYTE MODE** addition to ensure that each byte is examined individually, and not in pairs. The corresponding **IN CHARACTER MODE** addition is optional for processing character-type data objects.

## String Functions

Besides ABAP statements, predefined ABAP functions can be used to analyze character strings and byte strings. These functions can be divided in three groups: Descriptive functions (returning a numerical result), processing functions

(returning a character-type result), and predicate functions (returning a truth value). Predicate functions can only be used as conditions in control statements, as arguments of Boolean functions, or in relationships with Boolean operators.

Important descriptive string functions for character-type arguments can be used to determine the length of strings, to find character patterns in strings, or to count the occurrence of character patterns in strings:



	Result	
<code>strlen( `ABAP` )</code>	<b>7</b>	Length (incl. trailing blanks for strings)
<code>numofchar( `ABAP` )</code>	<b>5</b>	Length (excl. trailing blanks)
<code>find( val = `abapABAP` sub = `A` occ = 2 case = `X` ... ).</code>	<b>6</b>	Offset of pattern - second occurrence from left - case sensitive
<code>find_any_of( val = `abapAB` sub = `Ab` occ = -2 ... ).</code>	<b>1</b>	Offset of any character contained in pattern (always case sensitive) - second occurrence from right
<code>count( val = `abapABAP` sub = `A` case = `` ... ).</code>	<b>4</b>	Number of occurrences of pattern - not case sensitive
<code>count_any_of( val = `abapAB` sub = `Aa` off = 3 ... ).</code>	<b>1</b>	Number of occurrences of any character contained in pattern (always case sensitive ). - begin counting at offset = 3 from left

**Figure 90: Descriptive Functions for Character-Type Strings**

To determine the occupied length of a string, use the standard function STRLEN(). The corresponding function, XSTRLEN(), is available for byte-type data objects.

The following processing functions for character-type strings exist:



Result		
cmax( val1 = `ABAP` val2 = `ABCD` ... ). cmin( val1 = `ABAP` val2 = `ABCD` ... ).	`ABCD`  `ABAP`	Value of largest / smallest character-like argument
condense( val = ` AB AP ` del = ` ` ... ).	`AB AP`	Remove / replace characters
concat_lines_of( table = lt ... ).	row 1 row 2	Concatenate table rows to string
escape( val   = `://` format = 14 ).	`%3A%2F` `%2F`	Replace special characters with escape characters (constants in class CL_ABAP_FORMAT).
insert( val   = `AP` sub   = `BA` off   = 1 ).	`ABAP`	Insert one string into another string
match( val   = `BC402 Class` regex = `[\u]` occ   = -3 ).	`B`	Find occurrence of pattern defined by regular expression - upper case letter - third occurrence from right

Figure 91: Processing Functions for Character-Type Strings (1)



Result		
repeat( val1 = `A` occ = 3 ).	`AAA`	Multiply string
replace( val = `ABAP` sub = `A` with = `X` ... ).	`XBXP`	Replace characters
reverse( val1 = `ABAP` ).	`PABA`	Reverse string
segment( val   = `A AB ABC B` space = `AB` index = 3 ).	`C`	Segment delimited by any combination of characters defined by pattern - third segment for pattern AB
shift_left( val   = ` ABAP` places = 3 ). shift_right( val = ` ABAP` circular = 3 ).	`AP`  `BAP A`	Shift character string left / right
substring( val = `ABAPABAP` off = 3 len = 4 ).	`PABA`	Return substring

Figure 92: Processing Functions for Character-Type Strings (2)



Result	
<code>to_upper( val = `AbAp` ).</code>	<code>`ABAP`</code>
<code>to_lower( val = `AbAp` ).</code>	<code>`abap`</code>
<code>to_mixed( val = `TO_MIX` sep = `_` ... ).</code>	<code>`ToMix`</code>
<code>from_mixed( val = `FrMix` sep = `_` ... ).</code>	<code>`FR_MIX`</code>
<code>translate( val = `ABAP` from = `ABP` to = `al` ).</code>	<code>`ala`</code>
	Replace characters

Figure 93: Processing Functions for Character-Type Strings (3)

Finally, two predictive functions exist, `CONTAINS...()` and `MATCHES()`. These can be used to find out if a certain substring or search pattern is contained in a character string. `MATCHES()` accepts regular expressions only, while `CONTAINS...()` also supports strings defining search patterns. In addition, `CONTAINS...()` permits to check for multiple occurrences of search patterns.



Result	
<code>contains( val = `abapABAP` sub = `A` occ = 2 case = `X` ... ).</code>	<code>true</code>
<code>contains( val = `abapABAP` start = `Ap` occ = 2 off = 1 case = ` ` ... ).</code>	<code>false</code>
<code>contains_any_of( val = `ABAP` sub = `A` off = 2 occ = 1 ...).</code>	<code>true</code>
<code>matches( val = `abapABAP` regex = `[ABC]` off = 2 len = 4 case = `X` ).</code>	<code>true</code>
	Existence of pattern defined by regular expression - search for 'A', 'B', or 'C' - begin search at offset 2 - analyze 4 characters - case sensitive

Figure 94: Predictive Functions for Character-Type Strings

The code samples listed above represent variants used frequently. The functions may contain additional interface parameters not listed here. Depending on the function, the following interface parameters may be permitted:

- **val:** String to be analyzed.
- **sub:** Substring or search pattern.
- **regex:** Regular expression defining search pattern.
- **off, len:** Offset and length. Used to restrict operation on segment of string.
- **occ:** Number of matches that are to be searched for in the search range (integer value). Counted from right if negative.

If invalid values are assigned to the interface parameters, catchable exceptions are raised.

## String Expressions



**Hint:** The following section describes functionality available as of SAP NetWeaver 7.0 EhP 2.

A string expression formulates an operation with character-like operands. The result of a string expression is a character string. A string expression consists either of one string template or of two or more operands concatenated as a character string using the string operator **&&**. Each of the operands can itself be a string template.



**Note:** In operands with a fixed length, trailing blanks are not taken into account.

A **string template** is enclosed in two “|” characters. A string template defines a character string. The character string is determined by any sequence of the following syntax elements:

- Text literals
- Embedded expressions (data objects, calculated expressions, predefined functions, or function methods surrounded by { })
- Control characters

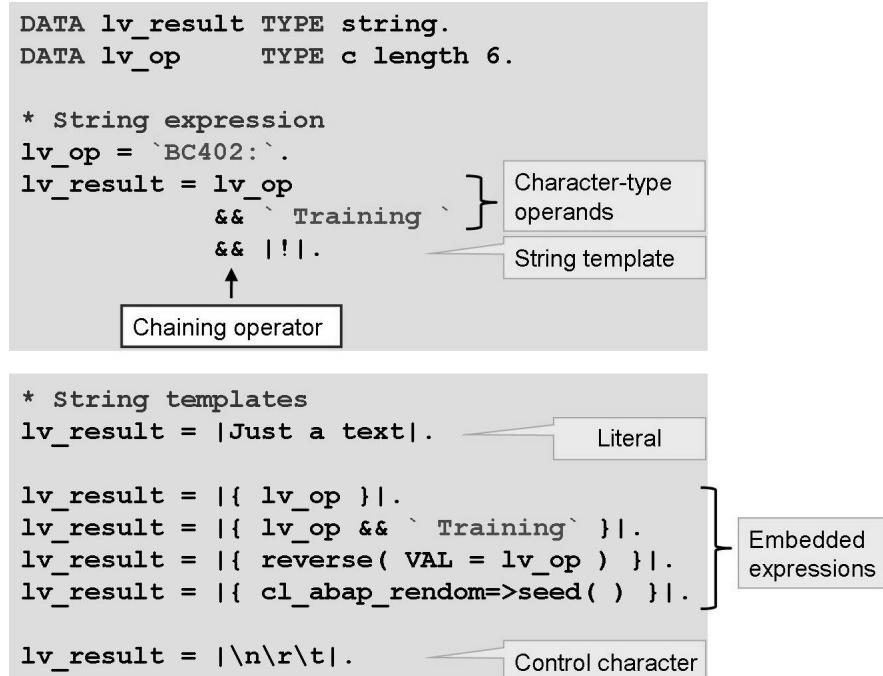


Figure 95: String Expressions / String Templates

→ **Note:** A string template that starts with “|” must be closed with “|” within the same line of source code. The only exceptions to this rule are line breaks in embedded expressions.

String templates can only be specified in Unicode programs.

The data type of the expression must be an elementary data type and the value of the expression must be character-type or must be convertible to a string. When a string template is analyzed, the value of each embedded expression is converted to a character string. By default, the string is formatted according to the data type of

the expression (predefined formatting). However, a number of formatting options can be used with the embedded expression. Formatting options have to be listed in the delimiting characters ( {, } ) behind the expression. The following options exist:



- **width**: Output length (decreasing minimum length is not possible).
- **align**: Output alignment (left, center, or right).
- **pad**: Replacement sign for excess spaces in output string.
- **case**: Letter format (upper case, lower case, or raw).
- **sign**: Visibility and position of sign in numerical type expressions.
- **exponent**: Exponent value for numerical expressions.
- **decimals**: Number of decimal places for numerical expressions.
- **zero**: Visibility of value 0 for numerical expressions (yes or no).
- **style**: Output style for numerical expressions (e.g. position of sign, usage of exponent).
- **currency**: Number of decimal spaces when formatting expressions of type i, p, or f. Valid values are defined in database table TCURC, column WAERS.
- **number**: Format of decimal and thousand separators for numerical expressions.
- **date**: Format for expressions of type d.
- **time**: Format for expressions of type t.
- **timestamp**: Format for expressions of type timestamp or timestamppl (both data elements defined in ABAP Dictionary).
- **timezone**: Time zone used to convert expressions of type timestamp or timestamppl. Valid values are defined in database table TTZZ, column TZONE.
- **country**: Country used to set format options **number**, **date**, **time**, and **timestamp** in one step. Valid values are defined in database table T005X, column LAND.



```

DATA lv_result TYPE string.

* Formatting embedded expressions
DO 5 TIMES.
  lv_result = |{ substring( VAL = sy-abcd
                            LEN = sy-index )
                WIDTH = sy-index + 8
                CASE = LOWER
                ALIGN = CENTER
                PAD = `.` }|.
  WRITE / lv_result.
ENDDO.

CONSTANTS lc_pi TYPE p DECIMALS 4 VALUE `3.1415`.

lv_result = |{ lc_pi STYLE = SCIENTIFIC
               EXPONENT = -2
               DECIMALS = 3 } |.
WRITE / lv_result.

```

....a....  
....ab....  
....abc....  
....abcd....  
....abcde....

314.150E-02

**Figure 96: Format Options for Embedded Expressions**

→ **Note:** Constants that may be assigned to the formatting parameters are defined in the class `CL_ABAP_FORMAT`.

## Regular Expressions



**Hint:** The following section describes functionality available as of SAP NetWeaver 7.0.

Patterns defined by regular expressions can be used in certain statements for character string processing or in string functions. A regular expression may qualify a single character, a character string, a search string, or a replacement string.

Regular expressions can be used in the statements FIND and REPLACE. In addition, the following string functions support the usage of regular expressions:

- COUNT ()
- COUNT\_... ()
- CONTAINS... ()
- FIND ()
- FIND\_... ()
- MATCH ()
- MATCHES ()
- REPLACE ()
- SUBSTRING ()
- SUBSTRING\_... ()

A regular expression is a literal containing all kinds of characters. However, some characters or character combinations have a special syntactical meaning.

To **describe a single character** by the regular expression, the following special character combinations may be used:



- \: Escape character for special characters.
- .: Placeholder for single character.
- \<c>: Placeholder for group of characters. Group depends on value of <c>. Example: \u = single upper case letter, \d = single digit
- [ ]: Value set for single character. Example: [ABC] = 'A', 'B', or 'C'
- [^ ]: Negation of a value set for single character. Example: [^ABC] = not ('A', 'B', or 'C')
- [ - ]: Value range for single characters. Example: [A-D] = any character between 'A' and 'D'
- [[:<c>:]]: Description for group of characters. Group depends on value of <c>. Example: [:upper:] = single upper case letter, [:digit:] = single digit



```
DATA lv_result TYPE string.

* Regular expressions - single character
lv_result = match( VAL      = `BC402abap`  

                   REGEX = `Cd` ) .
```

**BC402abap**

```
lv_result = match( VAL      = `BC402abap`  

                   REGEX = `C\d` ) .
```

**EC402abap**

```
lv_result = match( VAL      = `BC402abap`  

                   REGEX = `C[^7-9]` ) .
```

**BC402abap**

```
lv_result = match( VAL      = `BC402abap`  

                   REGEX = `.[12]` ) .
```

**BC402abap**

**Figure 97: Regular Expression - Single Character**

→ **Note:** To use one of the special characters (\, ^, [ ], .) in a regular expression, the escape character \ has to be put in front of the special character. Example: 'l' defines the beginning of a value set, '\l' defines the character [.]



**Hint:** The expression **[a-h][0-4]** defines a character string consisting of a lower case letter between 'a' and 'h' followed by any digit between 0 and 4 (e.g. 'a3'). In contrast, **[a-h0-4]** defines a character string consisting of one lower case letter between 'a' and 'h' **or** one digit between 0 and 4 (e.g. 'a' or '3').



To **describe a character string** by a regular expression, the following additional special character combinations may be used:

- **{n}**: Concatenation of **n** single characters.  
Example: '**A{5}**' = 'AAAAA'
- **{n,m}**: Concatenation of **n** to **m** single characters.  
Example: '**A{2,4}**' = 'AA', 'AAA', or 'AAAA'
- **?**: One or no single character. Thus **?** = **{0,1}**.  
Example: '**A?**' = 'A' or empty string
- **\***: Arbitrary number of characters (including empty string). Thus **\*** = **{0,}**.  
Example: '**A\***' = empty string, 'A', 'AA', 'AAA', ...
- **+**: Arbitrary number of characters (excluding empty string). Thus **\*** = **{1,}**.  
Example: '**A+**' = 'A', 'AA', 'AAA', ...
- **|**: Logical OR.  
Example: '**AB|CD**' = 'AB' or 'CD'
- **(?: )**: Subgroup definition (without registration). An operator following the character combination acts on subgroup.  
Example: '**AB(?:A|a)P**' = 'ABAP' or 'ABaP'  
Example: '**AB(?:AP){3}**' = 'ABAPAPAP'
- **( )**: Subgroup definition with registration. In addition to simple subgroup definition, the character patterns matching the subgroups are stored for latter reference. The matching pattern for subgroup **n** can be referred to using the character combination **\n**. In text replacements, the character combination **\$n** can be used to access the matching pattern.  
Example: '**(>|-)ABAP\1**' = '>ABAP>' or "-ABAP-".  
Example: '**(A|a)B\1P**' = 'ABAP' or 'aBaP'.



```
DATA lv_result TYPE string.
```

```
* Regular expressions - character string
lv_result = match( VAL      = `BC402abap`  
                  REGEX   = `BC{1,5}4+` ).
```

**BC402abap**

```
lv_result = match( VAL      = `BC402abap`  
                  REGEX   = `A*.{3}0?` ).
```

**BC402abap**

```
lv_result = match( VAL      = `BC402abap`  
                  REGEX   = `(\d.{5})|\1+` ).
```

**BC402abap**

```
lv_result = match( VAL      = `BC402abap`  
                  REGEX   = `(B|C).{4}\1`  
                  CASE    = space ).
```

**BC402abap**

**Figure 98: Regular Expression - Character String**

→ **Note:** To use one of the special characters ( {, }, ?, \*, +, |) in a regular expression, the escape character \ has to be put in front of the special character. Example: 'A\*' defines a pattern for a string consisting of an arbitrary occurrence of the character 'A'. In contrast, 'A\\*' defines the string 'A\*'.

One of the principle uses of regular expressions is the **search for substrings** in character strings. In general, a user is interested in a specific selection of character strings that match a regular expression.

In ABAP, the search of regular expressions is realized using the addition REGEX of the statement FIND, whereby the substrings found are determined without overlapping according to the **leftmost-longest rule**:

First, the substring is determined that is the furthest to the left in the character string, and that matches the regular expression (leftmost). If there are several substrings, the longest sequence is chosen (longest). This procedure is then repeated for the remaining sequence after the found location.



```
DATA lt_results TYPE string_table.
```

```
* Regular expressions - leftmost longest rule
FIND ALL OCCURRENCES OF
    regex 'A*BA+'
    IN 'ABAPPABAABAPPABA'
    RESULTS lt_results
```

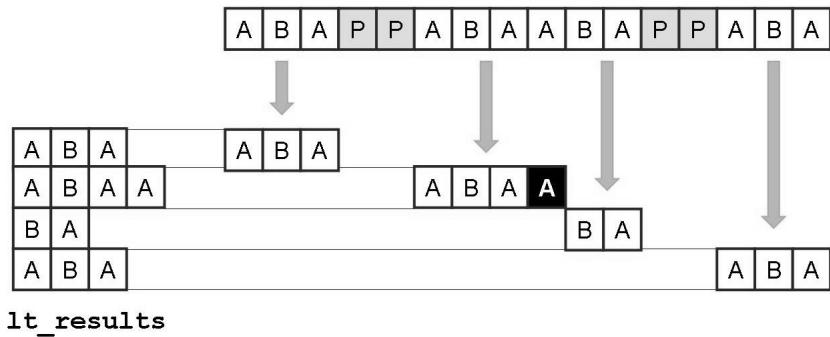


Figure 99: Regular Expression - Leftmost-Longest Rule

Special character combinations may be used to support searching in a character string:



- **^**: Anchor for offset before first character of line.

- **\$**: Anchor for offset after last character in a line.

- **\A**: Anchor for offset before first character of character string. If character string does not contain control characters, **\A** and **^** behave identically.

- **\z**: Anchor for offset after last character of character string. If character string does not contain control characters, **\z** and **\$** behave identically.

- **\<** or **\b**: Begin of word.

- **\>** or **\B**: End of word.

Example: '**(one)**' matched by 'one' but also by 'done' or 'lonely'. In contrast '**\<(one)\>**' is only matched by 'one'

- **(?= )**: Subsequent condition for previous regular expression. String has to match previous regular expression. In addition, the substring following matching position has to fulfil subsequent condition.

Example: '**(AB)(?=AP)**' is matched by 'ABAP\_BC402' since 'AB' is contained in literal and 'AP' is contained directly behind 'AB'

- **(?! )**: Subsequent condition for previous regular expression. String has to match previous regular expression. In addition, the substring following matching position must not fulfil subsequent condition.

Example: '**(AB)(?!AP)**' is not matched by 'ABAP\_BC402', the substring following 'AB' matches the subsequent condition.



```

DATA lv_string TYPE string.
DATA lt_result TYPE match_result_tab.

* Regular expressions - string pattern
FIND ALL OCCURRENCES OF REGEX reg
    IN lv_string RESULTS lt_result.

lv_string = `BC402abap BC402`.
```

- |    |                                      |                        |
|----|--------------------------------------|------------------------|
| 1) | <code>reg = 'BC402'</code>           | <b>BC402abap BC402</b> |
| 2) | <code>reg = '\&lt;BC402\&gt;'</code> | <b>BC402abap BC402</b> |
| 3) | <code>reg = '(BC402) (ab) '</code>   | <b>BC402abap BC402</b> |
| 4) | <code>reg = '(BC402) (?:ab) '</code> | <b>BC402abap BC402</b> |
| 5) | <code>reg = '(BC402) (?:ab) '</code> | <b>BC402abap BC402</b> |

**Figure 100: Regular Expression - Search Pattern**

**Note:** To use one of the special characters (`^`, `$`, `\`, `(`, `)`, `=`, `!`) in a regular expression, the escape character `\` has to be put in front of the special character. Example: `'A*'!` defines a pattern for a string consisting of an arbitrary occurrence of the character `'A'`. In contrast, `'A\*'!` defines the string `'A*'!`.

After searching, the **replacement of substrings** in character strings is the most important application of regular expressions. When replacing, the found locations of a search (or the substrings that match a regular expression), are replaced by one or more different character strings. In ABAP, the replacement is realized using regular expressions with the addition REGEX of the statement REPLACE.

In contrast to normal text replacements, when regular expressions are used, operators can be used in the replacement text that refer to the relevant found location. The following character combinations can be used in the replacement text:



- **\$`**: Complete text before current found location.
  - **\$`**: Complete text after current found location.
  - **\$n**: Character string stored in the register of subgroups for the n-th found location.
  - **\$&**: Complete text.



```
DATA lv_string TYPE string.  
DATA lv_result TYPE string.  
  
* Regular expressions - replace text  
replace( VAL = lv_string REGEX = reg WITH = rep ).  
  
lv_string = `More and more`.  
  
1)    reg = `and`    rep = `or`          More or more  
2)    reg = `and`    rep = `$``           More More more  
3)    reg = ` and`   rep = `$`           More more more  
4)    reg = `(.{4}).(and).(.*)`  
      rep = `$3,$3 $2 $3`            more, more and more  
5)    reg = `(More)`  
      rep = `$$`                   More and more and  
                                    more and more
```

Figure 101: Regular Expression - Special Characters for Replacement Text

→ **Note:** To use one of the special characters ( \$, ' , & , ) in a regular expression, the escape character \ has to be put in front of the special character. Example: '\$" defines a pattern related to the first subgroup. In contrast, '\$\'' defines the string' \$'.

## Logical Operators



	Character type	Byte
<b>Comparison operators</b>	CO, CA, CS, CP, CN, NA, NS, NP	BYTE-CO, BYTE-CA, BYTE-CS, BYTE-CN, BYTE-NA, BYTE-NS

```

DATA lv_cfield4 TYPE c
  LENGTH 4
  VALUE 'HUGO'.
DATA lv_cfield1 TYPE c
  LENGTH 1
  VALUE 'G'.

IF lv_field1 CA lv_field4.
  ...
ENDIF.

DATA lv_xfield3 TYPE x
  LENGTH 3
  VALUE '91B9A2'.
DATA lv_xfield1 TYPE x
  LENGTH 1
  VALUE 'B9'.

IF lv_xfield1 BYTE-CA lv_xfield3.
  ...
ENDIF.

```

**Figure 102: Lengths and Comparisons**

The following operators are available to compare the content of character-type data objects: CO, CN, CA, NA, CS, NS, CP, and NP. With the introduction of Unicode, corresponding operators with prefix BYTE- are available for byte-type data objects.

### Subfield Access

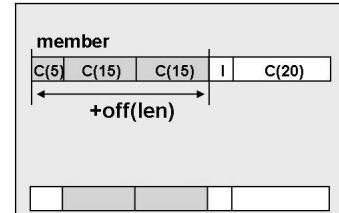
You can access a fragment of a character-type or byte-type data object directly by specifying the appropriate **offset** and **length**.



<statement> <structure>+<off>(<len>) ...

```
DATA: BEGIN OF member,
      title(5)    TYPE c,
      prename(15) TYPE c,
      surname(15) TYPE c,
      age         TYPE i,
      city(20)   TYPE c,
   END OF member.

PERFORM use_name USING member+5 (30).
```



- The structure must begin with a character fragment
- Access is only allowed within the initial character part
- The offset and length information are interpreted by character

**Figure 103: Access with Offset and Length Information for Elementary Data Objects**

The specifications for length and offset are interpreted as numbers of bytes for byte-type data objects, and as numbers of characters for character-type objects (that is, one or two bytes, depending on the system setting).

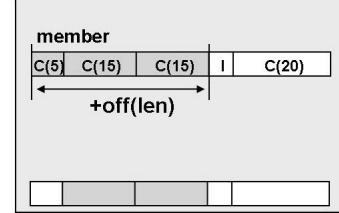
In flat structures, offset access is possible even if they are not completely character-type (that is, only the first field(s) are character-type), provided the access does not go beyond the character-type area.



<statement> <structure>+<off>(<len>) ...

```
DATA: BEGIN OF member,
      title(5)    TYPE c,
      prename(15) TYPE c,
      surname(15) TYPE c,
      age         TYPE i,
      city(20)   TYPE c,
   END OF member.

PERFORM use_name USING member+5 (30).
```



- The structure must begin with a character fragment
- Access is only allowed within the initial character part
- The offset and length information are interpreted by character

**Figure 104: Access with Offset and Length Specifications for Structures**



**Caution:** If possible, avoid using this method to access structure contents. Using components and string operations makes your program much more reliable and easier to read.

## Using Date and Time Fields

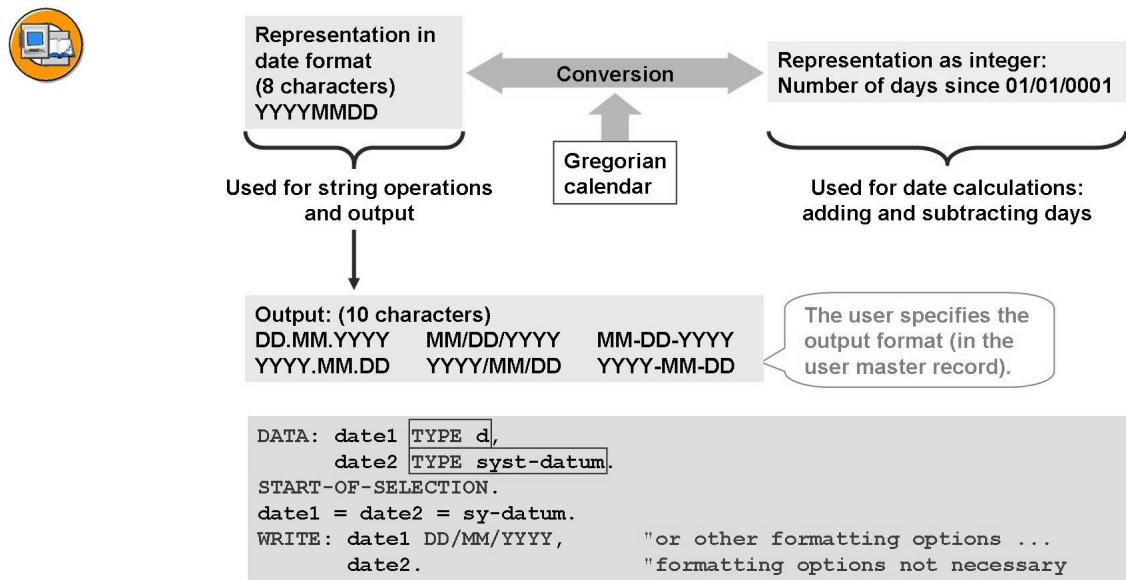


Figure 105: Date Fields

ABAP uses separate data types for dates and times. They are character-type data types that consist of 8 and 6 numeric characters, respectively. However, these data types also have a number of special attributes:

- The user-specific format is taken into account for input and output
- If a data object with type Date is used in arithmetic expressions or assigned to a numeric data object, its contents are converted to the number of days since January 1, 0001.
- If a data object with type Time is used in arithmetic expressions or assigned to a numeric data object, its contents are converted to the number of seconds since midnight.



**Caution:** You have to specify start values for date or time fields (in the VALUE addition of the DATA statement, for example) as **text literals**. This is because numeric literals have type Integer, which means they would be interpreted as the number of days since January 1, 0001 or the number of seconds since 12:00:00 a.m.



# Exercise 7: Processing Character Strings

## Exercise Objectives

After completing this exercise, you will be able to:

- Split strings
- Use conversion rules

## Business Example

In a sequential file records are available as long character strings. You want to extract these records and process them further in an appropriate format.

For simplicity's sake, the data in our exercise is provided by a function module to be called by the application you are developing. In practice, this module actually could read this data from a sequential file, which in turn might be generated by an external system.

We will limit the further processing of our data to outputting it in a list. In practice, you would probably write the data to the SAP system database.

In this exercise, we want to implement the first step for string processing. You will write a program that accepts the string, splits it into its components, and outputs them in structured form.

### Template:

BC402\_IDT\_DATA\_SET

### Solution:

BC402\_IDS\_DATA\_SET

### Task 1:

Copy program BC402\_IDT\_DATA\_SET to the name **ZBC402\_##\_DATA\_SET**, where ## is your group number. Analyze the program.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. What are the components of the program?

---

---

---

### Task 2:

Analyze and test function module BC402\_CREATE\_SEP\_STRING in the *Object Navigator*. Call the module in your program to provide a character string with the data to be processed. At first, use the default values for all the IMPORT parameters.

1. Open the function module in the *Object Navigator* and run a test. First leave all the IMPORT values set to their default values.  
Analyze the return value of the function module. What is the structure of the character string?

---

---

---

2. Familiarize yourself with the interface for function module BC402\_CREATE\_SEP\_STRING. How can use the function module interface to affect the behavior of the function module? What are the individual parameters used for?

---

---

---

3. Define a data object (suggested name: **gv\_datastring**) with data type string, in which your program accepts the string from the function module.
4. Call the function module BC402\_CREATE\_SEP\_STRING in the START-OF-SELECTION processing block. Use the default value for each IMPORT parameter. If the function module raises an exception, your program should return the termination message 038 from message class BC402.

*Continued on next page*

### Task 3:

Edit the character string. In the first step, remove the separators at the start and end of the character string.

1. Remove the separators at the start of the character string. Several options are available for doing so. You could use the SHIFT or REPLACE statements, for example, with the appropriate additions.



**Hint:** Refer to the keyword documentation to find out about the possible additions for these statements.

2. Remove the separators at the end of the character string. Again, use the SHIFT or REPLACE statement. You can also use the FIND statement to search for character sequence “##” and then use a length access to cut the string before the separator.

### Task 4:

Split the character string into (character-type) parts, so each part corresponds to the contents of one field. Use a structure consisting of character fields. Each field of the structure must be the same length as the corresponding component in the structure.

1. Define a structure type (suggested name: **ty\_s\_flight\_c**) and a structure (suggested name: **gs\_flight\_c**). Set type ty\_s\_flight\_c for both. The structure type should contain the following components:

#### Components of the Structure

Component name	Type	Length
mandt	e	3
carrid	e	3
connid	n	4
fldate	n	8
price	c	20
currency	e	5
planetype	e	10
seatsmax	n	10
seatsocc	n	10
paymentsum	c	22
seatsmax_b	n	10

*Continued on next page*

seatsocc_b	n	10
seatsmax_f	n	10
seatsocc_f	n	10



**Hint:** The copy template already contains a commented-out definition of the structure type, to save you from having to type it in.

2. Why can't components such as price, seatsmax, seatsocc, and so on, be set to numeric types?

---



---



---

3. Why can't the price component be set to type n?

---



---



---

### Task 5:

Convert the extracted data by copying it to data objects with suitable types. Refer to components with type SFLIGHT in the *ABAP Dictionary*. Again, create the target fields as components of a structure.



**Hint:** The type of a target field determines its display format. If you assign a source field to a target field of a different type, the runtime system converts it.

1. Define a structure type (suggested name: **ty\_s\_flight**) and a structure (suggested name: **gs\_flight**). Set type ty\_s\_flight for both.

Refer to component types with type SFLIGHT in the *ABAP Dictionary*. The structure type should consist of the following components:

*Continued on next page*

**Components of the Structure Type**

Component name	Type
carrid	sflight-carrid
connid	sflight-connid
fldate	sflight-fldate
price	sflight-price
currency	sflight-currency
planetype	sflight-planetype
seatsmax	sflight-seatsmax
seatsocc	sflight-seatsocc

2. Copy the identically named components of the character-type structure `gs_flight_c i` to the fields of the structure `gs_flight`. Note the conversion rule used by the system for the price component.

**Task 6:**

Then output the contents of structure `gs_flight` in a list.

1. Display the contents of the structure in a list. Use the appropriate formatting options for the `WRITE` statement for the `fldate` and `price` components.

## Solution 7: Processing Character Strings

### Task 1:

Copy program BC402\_IDT\_DATA\_SET to the name **ZBC402\_##\_DATA\_SET**, where ## is your group number. Analyze the program.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. What are the components of the program?

#### Answer:

The declaration of a local program structure type (flagged as comments).  
A text pool with a prepared list header.

### Task 2:

Analyze and test function module BC402\_CREATE\_SEP\_STRING in the *Object Navigator*. Call the module in your program to provide a character string with the data to be processed. At first, use the default values for all the IMPORT parameters.

1. Open the function module in the *Object Navigator* and run a test. First leave all the IMPORT values set to their default values.  
Analyze the return value of the function module. What is the structure of the character string?

**Answer:** The character string consists of several short character sequences, each separated by the “#” character. You also see double “#” characters at the start and end of the character string.

2. Familiarize yourself with the interface for function module BC402\_CREATE\_SEP\_STRING. How can use the function module interface to affect the behavior of the function module? What are the individual parameters used for?

#### Answer:

**im\_number:** The number of table entries to be read to make up the string  
**im\_table\_name:** The name of the table to read  
**im\_separator:** The separator for each individual field in the record  
**im\_unique:** Are identical records allowed?

*Continued on next page*

3. Define a data object (suggested name: **gv\_datastring**) with data type string, in which your program accepts the string from the function module.
  - a) See the source code extract from the model solution.
4. Call the function module BC402\_CREATE\_SEP\_STRING in the START-OF-SELECTION processing block. Use the default value for each IMPORT parameter. If the function module raises an exception, your program should return the termination message 038 from message class BC402.
  - a) See the source code excerpt from the model solution.

### Task 3:

Edit the character string. In the first step, remove the separators at the start and end of the character string.

1. Remove the separators at the start of the character string. Several options are available for doing so. You could use the SHIFT or REPLACE statements, for example, with the appropriate additions.



**Hint:** Refer to the keyword documentation to find out about the possible additions for these statements.

- a) See the source code excerpt from the model solution.
2. Remove the separators at the end of the character string. Again, use the SHIFT or REPLACE statement. You can also use the FIND statement to search for character sequence “##” and then use a length access to cut the string before the separator.
  - a) See the source code excerpt from the model solution.

### Task 4:

Split the character string into (character-type) parts, so each part corresponds to the contents of one field. Use a structure consisting of character fields. Each field of the structure must be the same length as the corresponding component in the structure.

1. Define a structure type (suggested name: **ty\_s\_flight\_c**) and a structure (suggested name: **gs\_flight\_c**). Set type ty\_s\_flight\_c for both. The structure type should contain the following components:

*Continued on next page*

### Components of the Structure

Component name	Type	Length
mandt	e	3
carrid	e	3
connid	n	4
fldate	n	8
price	c	20
currency	e	5
planetype	e	10
seatsmax	n	10
seatsocc	n	10
paymentsum	c	22
seatsmax_b	n	10
seatsocc_b	n	10
seatsmax_f	n	10
seatsocc_f	n	10



**Hint:** The copy template already contains a commented-out definition of the structure type, to save you from having to type it in.

- a) See the source code excerpt from the model solution.
- 2. Why can't components such as price, seatsmax, seatsocc, and so on, be set to numeric types?

**Answer:** When you split the character string, this results in character-type parts at first. You have to convert them to numeric data types in a second step.

- 3. Why can't the price component be set to type n?

**Answer:** The corresponding parts also contain decimal points. But the decimal point is not in the value range of type n.

*Continued on next page*

## Task 5:

Convert the extracted data by copying it to data objects with suitable types. Refer to components with type SFLIGHT in the *ABAP Dictionary*. Again, create the target fields as components of a structure.



**Hint:** The type of a target field determines its display format. If you assign a source field to a target field of a different type, the runtime system converts it.

1. Define a structure type (suggested name: **ty\_s\_flight**) and a structure (suggested name: **gs\_flight**). Set type ty\_s\_flight for both.

Refer to component types with type SFLIGHT in the *ABAP Dictionary*. The structure type should consist of the following components:

### Components of the Structure Type

Component name	Type
carrid	sflight-carrid
connid	sflight-connid
fldate	sflight-fldate
price	sflight-price
currency	sflight-currency
planetype	sflight-planetype
seatsmax	sflight-seatsmax
seatsocc	sflight-seatsocc

2. a) See the source code excerpt from the model solution.  
Copy the identically named components of the character-type structure gs\_flight\_c\_i to the fields of the structure gs\_flight. Note the conversion rule used by the system for the price component.

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

## Task 6:

Then output the contents of structure gs\_flight in a list.

1. Display the contents of the structure in a list. Use the appropriate formatting options for the WRITE statement for the fldate and price components.  
a) See the source code excerpt from the model solution.

*Continued on next page*

## Result

Source code excerpt from the model solution:

```

REPORT  bc402_ids_data_set MESSAGE-ID bc402.

TYPES:
  BEGIN OF ty_s_flight_c,
    mandt      TYPE c LENGTH 3,
    carrid     TYPE c LENGTH 3,
    connid     TYPE n LENGTH 4,
    fldate     TYPE n LENGTH 8,
    price      TYPE c LENGTH 20,
    currency   TYPE c LENGTH 5,
    planetype  TYPE c LENGTH 10,
    seatsmax   TYPE n LENGTH 10,
    seatsocc   TYPE n LENGTH 10,
    paymentsum TYPE c LENGTH 22,
    seatsmax_b TYPE n LENGTH 10,
    seatsocc_b TYPE n LENGTH 10,
    seatsmax_f TYPE n LENGTH 10,
    seatsocc_f TYPE n LENGTH 10,
  END OF ty_s_flight_c,

  BEGIN OF ty_s_flight,
    carrid     TYPE sflight-carrid,
    connid     TYPE sflight-connid,
    fldate     TYPE sflight-fldate,
    price      TYPE sflight-price,
    currency   TYPE sflight-currency,
    planetype  TYPE sflight-planetyp,
    seatsmax   TYPE sflight-seatsmax,
    seatsocc   TYPE sflight-seatsocc,
  END OF ty_s_flight.

DATA:
  gv_datastring  TYPE string,
  gv_set_string  TYPE string,
  gv_offset      TYPE i,
  gs_flight_c    TYPE ty_s_flight_c,
  gs_flight      TYPE ty_s_flight.

START-OF-SELECTION.

* retrieve character string with data

```

*Continued on next page*

```

CALL FUNCTION 'BC402_CREATE_SEP_STRING'
*      EXPORTING
*          im_number      = '1'
*          im_table_name = 'SFLIGHT'
*          im_separator  = '#'
*          im_unique     = 'X'
      IMPORTING
          ex_string      = gv_datastring
      EXCEPTIONS
          no_data        = 1
          OTHERS         = 2.
IF sy-subrc <> 0.
MESSAGE a038.
ENDIF.

* remove leading and trailing separators

* solution 1 - SHIFT, FIND & Offset access
gv_set_string = gv_datastring.
SHIFT gv_set_string BY 2 PLACES.
FIND '##' IN gv_set_string
    MATCH OFFSET gv_offset.
IF sy-subrc = 0.
    gv_set_string = gv_set_string(gv_offset).
ENDIF.

* solution 2 - REPLACE
* gv_set_string = gv_datastring.
* REPLACE ALL OCCURRENCES OF '##'
*     IN gv_set_string WITH ''.

* solution 3 - SHIFT CIRCULAR & SHIFT
* gv_set_string = gv_datastring.
* SHIFT gv_set_string RIGHT CIRCULAR BY 2 PLACES.
* SHIFT gv_set_string LEFT BY 4 PLACES.
*
* solution 4 - SHIFT DELETING
* gv_set_string = gv_datastring.
* SHIFT gv_set_string LEFT DELETING LEADING '#'.
* SHIFT gv_set_string RIGHT DELETING TRAILING '#'.
* SHIFT gv_set_string LEFT DELETING LEADING ''.

* split into (charlike) fragments corresponding to components
SPLIT gv_set_string AT '#' INTO
    gs_flight_c-mandt

```

*Continued on next page*

```
gs_flight_c-carrid  
gs_flight_c-connid  
gs_flight_c-fldate  
gs_flight_c-price  
gs_flight_c-currency  
gs_flight_c-planetype  
gs_flight_c-seatsmax  
gs_flight_c-seatsocc  
gs_flight_c-paymentsum  
gs_flight_c-seatsmax_b  
gs_flight_c-seatsocc_b  
gs_flight_c-seatsmax_f  
gs_flight_c-seatsocc_f.  
  
* convert fragments into proper data types  
MOVE-CORRESPONDING gs_flight_c TO gs_flight.  
  
* output result  
WRITE: /  
      gs_flight-carrid,  
      gs_flight-connid,  
      gs_flight-fldate DD/MM/YYYY,  
      gs_flight-price CURRENCY gs_flight-currency,  
      gs_flight-currency,  
      gs_flight-planetype,  
      gs_flight-seatsmax,  
      gs_flight-seatsocc.
```

## Exercise 8: Splitting Character Strings in Internal Tables

### Exercise Objectives

After completing this exercise, you will be able to:

- Define one-column tables with data type STRING
- Split strings into internal tables
- Process internal tables in a loop

### Business Example

The character string from the exercise on character string processing will now contain more than one data record.

Enhance your program to save each record in the string in a separate line of an internal table. From this table, you will then separate the individual records into their elementary components and output them as before.

#### Template:

BC402\_IDS\_DATA\_SET

#### Solution:

BC402\_IDS\_SPLIT\_ITAB

### Task 1:

Copy program BC402\_IDS\_DATA\_SET (or your own program, ZBC402\_##\_DATA\_SET) to the name **ZBC402\_##\_SPLIT\_ITAB**, where ## is your group number. Make sure the string contains more than one record.

1. Copy the program and all its subcomponents.
2. Use function module BC402\_CREATE\_SEP\_STRING to generate a string of 30 records. To do so, evaluate parameter IM\_NUMBER with an appropriately typed constant (suggested name: C\_NUMBER).

*Continued on next page*

## Task 2:

After removing the separators at the start and end of the character string, break it down into parts, each of which corresponds to a single record. Use the option of specifying an internal table with an elementary, character-type line type as the target of the SPLIT statement.

1. Check (in the *debugger*, for example) whether the technique used in the program for removing the separators at the start and end of the string works even if the string contains multiple records. What do you have to take into account here?  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
2. Define a single-column internal table with standard table type. The lines of the internal table should have data type STRING (suggested name: IT\_SETS).
3. Implement a new SPLIT statement directly before the existing SPLIT statement in which you fill the internal table with the partial character strings. Split the character string so that each partial string corresponds to exactly one record.
4. Make sure that the split into individual records, type conversion, and output of the components occurs in a consecutive loop for all lines in the internal table.

## Solution 8: Splitting Character Strings in Internal Tables

### Task 1:

Copy program BC402\_IDS\_DATA\_SET (or your own program, ZBC402\_##\_DATA\_SET) to the name **ZBC402\_##\_SPLIT\_ITAB**, where ## is your group number. Make sure the string contains more than one record.

1. Copy the program and all its subcomponents.
  - a) Carry out this step as usual.
2. Use function module BC402\_CREATE\_SEP\_STRING to generate a string of 30 records. To do so, valuate parameter IM\_NUMBER with an appropriately typed constant (suggested name: C\_NUMBER).
  - a) See the source code excerpt from the model solution.

### Task 2:

After removing the separators at the start and end of the character string, break it down into parts, each of which corresponds to a single record. Use the option of specifying an internal table with an elementary, character-type line type as the target of the SPLIT statement.

1. Check (in the *debugger*, for example) whether the technique used in the program for removing the separators at the start and end of the string works even if the string contains multiple records. What do you have to take into account here?

**Answer:** When multiple records are present, they are separated by the “##” character combination. This can result in problems when you search for this pattern with the FIND or REPLACE statements.

2. Define a single-column internal table with standard table type. The lines of the internal table should have data type STRING (suggested name: IT\_SETS).
  - a) See the source code excerpt from the model solution.
3. Implement a new SPLIT statement directly before the existing SPLIT statement in which you fill the internal table with the partial character strings. Split the character string so that each partial string corresponds to exactly one record.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

4. Make sure that the split into individual records, type conversion, and output of the components occurs in a consecutive loop for all lines in the internal table.
- a) See source code excerpt from the model solution

## Result

Source code excerpt from the model solution:

```
REPORT bc402_ids_split_itab MESSAGE-ID bc402.
```

**TYPES:**

```
BEGIN OF ty_s_flight_c,
  mandt      TYPE c LENGTH 3,
  carrid     TYPE c LENGTH 3,
  connid     TYPE n LENGTH 4,
  fldate     TYPE n LENGTH 8,
  price       TYPE c LENGTH 20,
  currency    TYPE c LENGTH 5,
  planetype   TYPE c LENGTH 10,
  seatsmax    TYPE n LENGTH 10,
  seatsocc    TYPE n LENGTH 10,
  paymentsum  TYPE c LENGTH 22,
  seatsmax_b  TYPE n LENGTH 10,
  seatsocc_b  TYPE n LENGTH 10,
  seatsmax_f  TYPE n LENGTH 10,
  seatsocc_f  TYPE n LENGTH 10,
END OF ty_s_flight_c,

BEGIN OF ty_s_flight,
  carrid     TYPE sflight-carrid,
  connid     TYPE sflight-connid,
  fldate     TYPE sflight-fldate,
  price       TYPE sflight-price,
  currency    TYPE sflight-currency,
  planetype   TYPE sflight-planetype,
  seatsmax    TYPE sflight-seatsmax,
  seatsocc    TYPE sflight-seatsocc,
END OF ty_s_flight.
```

```
CONSTANTS c_number TYPE i VALUE 30.
```

**DATA:**

```
gv_datastring  TYPE string,
gv_set_string  TYPE string,
```

*Continued on next page*

```

gs_flight_c      TYPE ty_s_flight_c,
gs_flight        TYPE ty_s_flight.

DATA:
gt_sets TYPE STANDARD TABLE OF string
WITH NON-UNIQUE DEFAULT KEY.

START-OF-SELECTION.

* retrieve character string with data (several data sets this time)
CALL FUNCTION 'BC402_CREATE_SEP_STRING'
EXPORTING
im_number      = c_number
*           im_table_name = 'SFLIGHT'
*           im_separator  = '#'
*           im_unique     = 'X'
IMPORTING
ex_string      = gv_datastring
EXCEPTIONS
no_data        = 1
OTHERS          = 2.
IF sy-subrc <> 0.
MESSAGE a038.
ENDIF.

* remove leading and trailing separators

* solution 1 - doesn't work here
*   gv_set_string = gv_datastring.
*   SHIFT gv_set_string BY 2 PLACES.
*   FIND '##' IN gv_set_string
*       MATCH OFFSET gv_offset.
*   IF sy-subrc = 0.
*       gv_set_string = gv_set_string(gv_offset).
*   ENDIF.

* solution 2 - doesn't work here
*   gv_set_string = gv_datastring.
*   REPLACE ALL OCCURRENCES OF '##'
*       IN gv_set_string WITH ''.

* solution 3 - works fine
gv_set_string = gv_datastring.

```

*Continued on next page*

```

SHIFT gv_set_string RIGHT CIRCULAR BY 2 PLACES.
SHIFT gv_set_string LEFT BY 4 PLACES.

* solution 4 - works fine
* gv_set_string = gv_datastring.
* SHIFT gv_set_string LEFT DELETING LEADING '#'.
* SHIFT gv_set_string RIGHT DELETING TRAILING '#'.
* SHIFT gv_set_string LEFT DELETING LEADING ''.

* split into fragments, each corresponding to one data set

SPLIT gv_datastring AT '##' INTO TABLE gt_sets.

LOOP AT gt_sets INTO gv_set_string.

* split into (charlike) fragments corresponding to components
  SPLIT gv_set_string AT '#' INTO
    gs_flight_c-mandt
    gs_flight_c-carrid
    gs_flight_c-connid
    gs_flight_c-fldate
    gs_flight_c-price
    gs_flight_c-currency
    gs_flight_c-planetype
    gs_flight_c-seatsmax
    gs_flight_c-seatsocc
    gs_flight_c-paymentsum
    gs_flight_c-seatsmax_b
    gs_flight_c-seatsocc_b
    gs_flight_c-seatsmax_f
    gs_flight_c-seatsocc_f.

* convert fragments into proper data types
  MOVE-CORRESPONDING gs_flight_c TO gs_flight.

* output result
  WRITE: /
    gs_flight-carrid,
    gs_flight-connid,
    gs_flight-fldate DD/MM/YYYY,
    gs_flight-price CURRENCY gs_flight-currency,
    gs_flight-currency,
    gs_flight-planetype,
    gs_flight-seatsmax,
    gs_flight-seatsocc.

```

*Continued on next page*

ENDLOOP.



## Lesson Summary

You should now be able to:

- Understand Unicode and the stricter syntax requirements it imposes
- Use operators and functions to analyze byte strings and character strings
- Use string templates to define character strings
- Use regular expressions to define search patterns

# Lesson: Table Types and Their Use

## Lesson Overview

This lesson deals with internal tables. You learn about the differences between the three table types, as well as what you have to watch out for when using these tables. The difference between index access and key access is crucial. As such, you also learn how to differentiate index accesses from key accesses.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the differences between standard, sorted, and hashed table types
- Tell the difference between key accesses and index accesses to internal tables
- Use the different table types correctly

## Business Example

You develop applications in which large volumes of data are stored and processed in internal tables. You have heard that sorted tables and hashed tables can vastly improve performance. You therefore want to learn how to use these table types.

## Table Types - Standard, Sorted, and Hashed

As you know, you have to select one of three table types when defining internal tables and table categories: standard tables, sorted tables, and hashed tables. The table type determines how data in the table is stored and accessed.



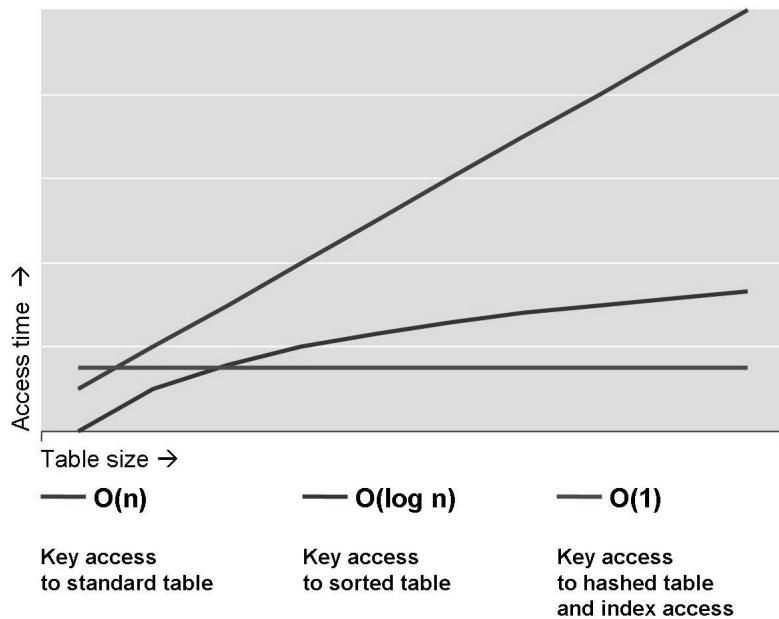
Table kind	Index tables		Hashed table
	STANDARD TABLE	SORTED TABLE	HASHED TABLE
Index access 	✓	✓	
Key access 	✓	✓	✓
Uniqueness	NON-UNIQUE	UNIQUE   NON-UNIQUE	UNIQUE
Access			 Only using key
	Primarily using index	Primarily using key	

**Figure 106: Link Between Table Type and Access Type**

Storage and access are most simple for standard tables. From the functionality aspect, the other table types hardly offer any additional features (aside from the option of unique keys, which we discuss below). To the contrary: some access types that use indexes are not suitable or even forbidden for sorted and hashed tables.

But the major motivation for using sorted and hashed tables is not the additional features, but rather in the runtime requirements for accessing large internal tables. (An exception to this is the selection of a unique key for sorted tables – see the last section of this lesson for more information).

The diagram below schematically shows how the access times for key access (full qualified key) of the various table types vary with table size:



**Figure 107: Access Time for Internal Tables**

While the access times are hardly different at all when tables are small, the more lines a table contains, the larger the differences become. A remarkable feature is that the access times for key access to hashed tables are not dependent on the table size.

In the sections below, you see how data is stored in the different table types and how this results in the large runtime differences during key access.

### Standard Tables

Standard tables are managed internally using a table index (=line numbers). New lines are either appended at the end of the table or inserted at specific positions.

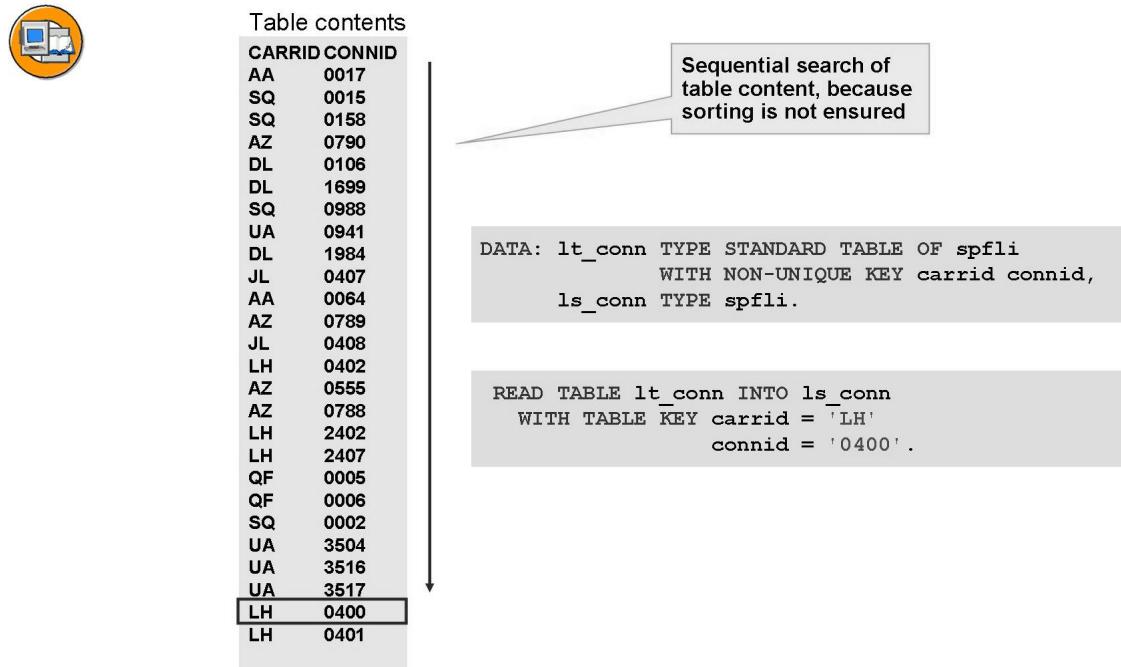


Figure 108: Sequential Search for Key Access to Standard Table

Since the lines within the internal table can have any sequence, the runtime environment has to **search the entire table sequentially** during a key access.. While this can be done relatively quickly for small tables, the search effort increases linearly with the number of lines: twice as many lines means a twice as long mean access time.

→ **Note:** If the internal table is sorted explicitly (with the SORT statement), the BINARY SEARCH addition can speed up the access significantly. This option is discussed in a later lesson.

## Sorted Tables

Like standard tables, sorted tables are also managed using a table index. But their entries are always sorted in ascending order by the table key. As a result, the runtime environment can automatically perform a **binary search** during table key accesses.

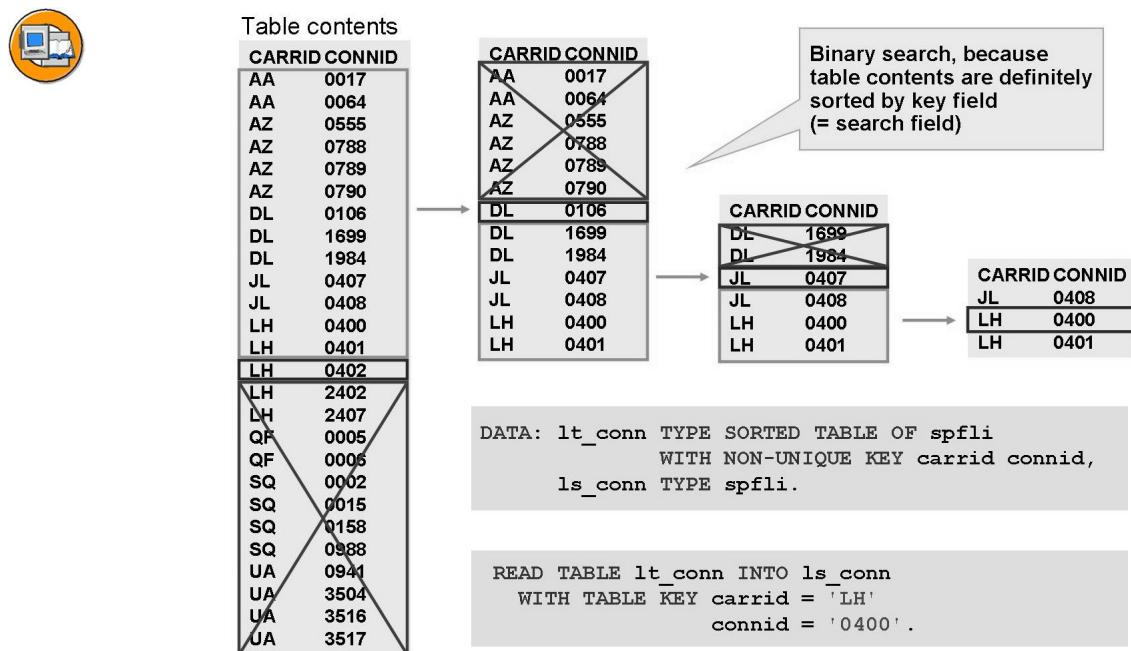


Figure 109: Binary Search for Key Access to Standard Table

Based on the total number of lines, the runtime environment determines an index in the middle and reads a line by index access there. By comparing the key field values of the line with the search criteria, it can determine whether the sought entry lies before or after the current line. This halves the data volume to be searched in a single step. This step is repeated for the remaining lines until the sought entry is found (or the system determines that no entry in the table matches the criteria).



**Hint:** If the sorted table has a **non-unique key**, several lines could match the criterion. As a result of sorting, these lines are directly adjacent in a block. The runtime environment always returns the **first** line of the block, that is, the one with the smallest index.

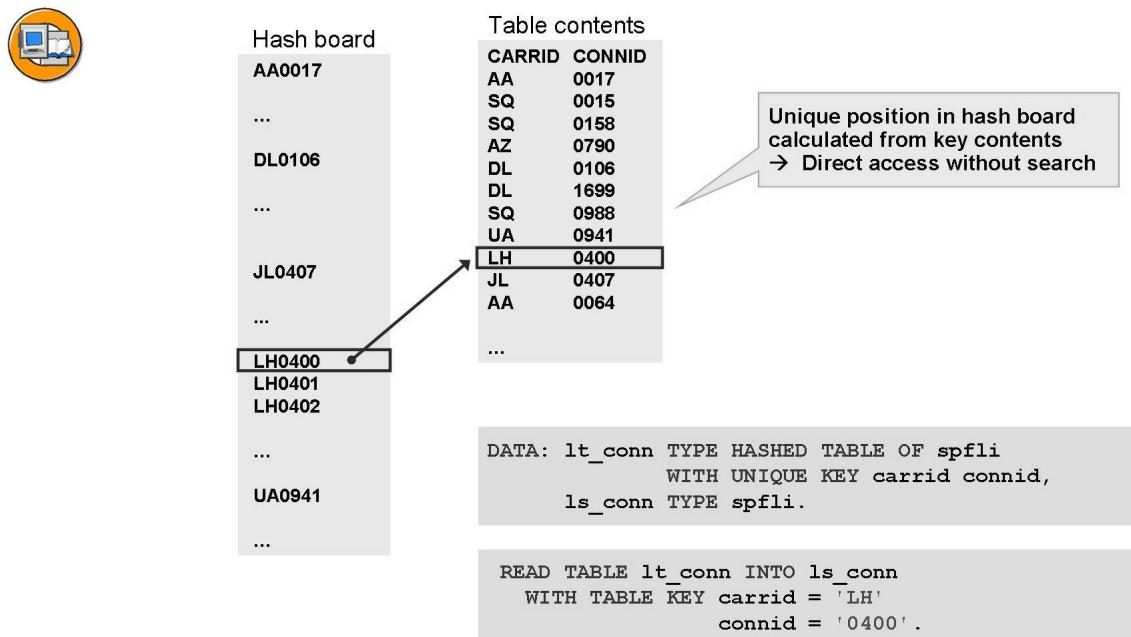
The sort sequence of a sorted table must never be disturbed – even temporarily. In many cases, the runtime environment automatically ensures that the table contents are sorted correctly – for example, when a sorted table is specified for a database access after INTO TABLE or when a MOVE statement copies the contents of a table into a sorted table. The runtime environment also ensures the correct sort sequence for key accesses.



**Caution:** When modifying index accesses are involved, it is up to the developer to ensure the sort sequence is retained. If the sort sequence is destroyed, an **ITAB\_ILLEGAL\_SORT\_ORDER** runtime error, which cannot be handled, occurs.

## Hashed Tables

Hashed tables are managed internally using a **hash algorithm**. To enable this, a **hash board** is created in addition to the actual table and the memory addresses of the table lines are managed in that table. While the table contents themselves are completely unsorted, a hash function lets you calculate the position of the entry in the hash board directly from the key values. Accordingly, the key of a hashed table must always be unique.



**Figure 110: Access Using Hash Board for Hashed Tables**

The effort required to calculate the position in the hash board and then read the table line is identical for all lines, regardless of the number of table entries.

The entries in the hash board are not consecutive, but instead spread over a certain memory area. These gaps make it possible to insert additional lines later. The hash function used determines how large the gaps are. If the runtime environment calculates that the entries in the hash board are too close to one another when inserting a line, the hash board is reorganized. This means calculating the positions with a different hash function that distributes the entries over a larger memory area.



**Hint:** Loops (LOOP ... ENDLOOP) ignore the hash board and search the table contents directly. The content is unsorted, but can be sorted for this purpose with the SORT statement.



## Index Accesses and Key Accesses

When using the different table types, the differentiation between index accesses and key accesses is crucial.

Index accesses are not allowed for hashed tables. Furthermore, you must be cautious when using modifying index accesses of sorted tables, as you run the risk of violating the sort sequence, which causes a runtime error.

	Standard table	Sorted table	Hashed table
SORT	✓	🚫	✓
Index-Zugriff	➡ n		
APPEND	✓	⚡	🚫
INSERT	✓	⚡	🚫
READ TABLE	✓	⚡	🚫
MODIFY	✓	⚡	🚫
DELETE	✓	✓	🚫
LOOP AT	✓	✓	🚫
Key access	🔑	Append at end of table	
INSERT	✓	✓	✓
READ TABLE	✓	✓	✓
MODIFY	✓	✓	✓
DELETE	✓	✓	✓
LOOP AT	✓	✓	✓

Figure 111: Overview of the Basic Operations

While the syntax check ensures no index accesses are made of hashed tables, the developer of sorted tables has to be able to spot modifying index accesses, and avoid their incorrect use. This is described in the next sections.

### Single Record Read Accesses

Single records from internal tables are read using the READ TABLE statement.



```
DATA: lt_conn TYPE STANDARD TABLE OF spfli
      WITH NON-UNIQUE KEY carrid connid,
      ls_conn LIKE LINE OF lt_conn.
```

Read by index

```
READ TABLE lt_conn
      INTO ls_conn
      INDEX 1
      TRANSPORTING cityfrom.
```

**Not allowed  
for hashed**

Read by (explicit) table key

```
READ TABLE lt_conn
      INTO ls_conn
      WITH TABLE KEY
            carrid = 'LH'
            connid = '0400'
      TRANSPORTING cityfrom..
```

Read by (implicit) table key

```
READ TABLE lt_conn
      FROM ls_conn
      INTO ls_conn
      TRANSPORTING cityfrom..
```

**Implicit specification  
of key as content of  
structure**

Read by any key (explicit)

```
READ TABLE lt_conn
      INTO ls_conn
      WITH KEY
            connid = '0400'
      TRANSPORTING cityfrom..
```

**With READ by index:  
No problems with sorted tables**

**Figure 112: Reading by Index and Key**

Index accesses are characterized by the INDEX addition, which specifies an explicit index. This addition is not allowed for hashed tables.

For key accesses, we differentiate between implicit specification the key with the FROM addition and explicit specification using WITH (TABLE) KEY.

### Key is implicit

When the key is specified implicitly, the key field values are taken from the structure after **FROM**. FROM and INTO do not need to have the same structure.

### Key is explicit

When the key is specified explicitly, we differentiate between specifying the table key using **WITH TABLE KEY** and specifying any other key using the addition **WITH KEY**.

If you use the **WITH TABLE KEY** addition, the syntax check demands that all the key fields in the table are listed and supplied with a value. If a key field is missing, this causes a syntax error.

If you use the **WITH KEY** addition, you can specify any column identifier. The syntax check does not compare them with the key fields.



**Hint:** The addition does not have a direct impact on runtime. When you access sorted and hashed tables, the runtime environment uses the binary search or hash algorithm, respectively, even if you use the **WITH KEY** addition. The only prerequisite is that all key fields must be specified. The **WITH TABLE KEY** addition can help make a program more robust, however: It can prevent a key access from dropping in performance because the definition of the table key was changed, for example, and saves you from having to modify the access syntax.

The optional **TRANSPORTING** addition lets you copy selected columns into the specified structure (after **INTO**) instead of the entire line.



**Hint:** If you use the special **TRANSPORTING NO FIELDS** addition, nothing is read at all. Accordingly, **INTO** is not needed. The runtime environment simply looks for a line with the specified index and/or key, setting the **sy-subrc** and **sy-tabix** system fields accordingly. It gives you a high-performance way to check whether or not the table contains a specific line.

### Single Record Write Access

We differentiate modifying access to records in internal tables depending on whether a new entry is to be added, an existing entry changed, or a single entry deleted. The differences between index access and key access are particularly important here, because modifying index accesses of sorted tables can cause runtime errors.

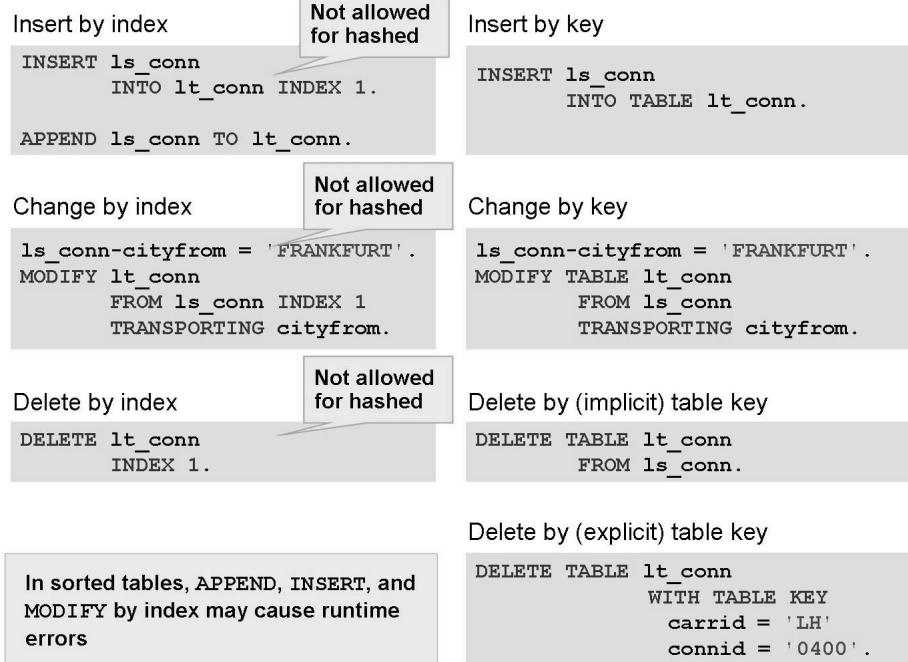


Figure 113: Modifying Index and Key Accesses



**Hint:** In all modifying single record accesses **by key**, the keyword **TABLE** appears next to the name of the internal table (that is, **INTO TABLE**, **MODIFY TABLE**, **DELETE TABLE**). This addition is not used in modifying single record accesses **by index** (that is, **INTO lt\_conn**, **MODIFY lt\_conn**, **DELETE lt\_conn**). Therefore, **INDEX** and **TABLE** are mutually exclusive.

Note the following details for the individual statements:

#### DELETE by key

Table key either implicit (FROM) or explicit (WITH TABLE KEY). WITH KEY is not supported (see the DELETE statement with WHERE condition).

Depending on the table type, the runtime environment searches for the line to be deleted sequentially, binary, or using a hash algorithm.

#### MODIFY by key

Only implicit key possible. Depending on the table type, the runtime environment determines line to be changed sequentially, binary, or using a hash algorithm.

Only non-key fields are changed. The explicit specification of key fields after TRANSPORTING causes a syntax error with sorted and hashed tables.

## INSERT by key

Only implicit key possible.

For **hashed tables**, the line is appended to the table contents, an entry is added to the hash board, and the table is reorganized if necessary. For sorted tables, a binary search determines the insert position. For standard tables, the line is simply added to the end.

## APPEND and INSERT by index



**Caution:** Allowed for sorted tables, but only if you are sure the insert position is correct and the sort sequence is retained. When in doubt, insert by key access.

## MODIFY by index



**Caution:** This statement also allows changes to key fields. This is dangerous for sorted tables, as it potentially destroys the search sequence.

You can use the optional **TRANSPORTING** addition with MODIFY similar to READ TABLE described above: instead of the entire structure, only selected components are copied to the desired line in the internal table.



```
DATA: lt_conn TYPE STANDARD TABLE OF spfli
      WITH NON-UNIQUE KEY carrid connid,
      ls_conn LIKE LINE OF lt_conn.
```

```
LOOP AT lt_conn INTO ls_conn
  WHERE carrid = 'LH' AND
        connid = '0400'.

  * add new line at current position
  ls_conn-carrid = 'LH'.
  ls_conn-connid = '0399'.
  INSERT ls_conn INTO lt_conn.

  * modify current line
  ls_conn-cityfrom = 'FRANKFURT'.
  MODIFY lt_conn FROM ls_conn
    TRANSPORTING cityfrom.

  * delete current line
  DELETE lt_conn.

ENDLOOP.
```

INSERT at the current position (INDEX sy-tabix)

MODIFY the current row (INDEX sy-tabix)

DELETE the current row (INDEX sy-tabix)

Abbreviated syntax only sensible in loops, but does not trigger syntax errors outside loops.  
But it causes a runtime error!

Figure 114: Abbreviated Syntax for Index Access in Loops

When you make modifying single record accesses by index, you can omit the explicit index specification (see diagram above). In this case, the runtime system implicitly uses the current value from sy-tabix. These syntax variants are only sensible within loops of the corresponding internal table and can only be used there.



**Caution:** The constraint that these statements can only be used within loops is **not verified by the syntax check**. If you use them outside of loops, it causes a **fatal exception (TABLE\_ILLEGAL\_STATEMENT)**.

### Key Access with WHERE Condition

You should be familiar with restrictions using WHERE from processing internal tables in LOOPS. WHERE conditions can also be specified in special variants of MODIFY and DELETE:



```
DATA: lt_conn TYPE STANDARD TABLE OF spfli
      WITH NON-UNIQUE KEY carrid connid,
      ls_conn LIKE LINE OF lt_conn.
```

Read

```
LOOP AT lt_conn INTO ls_conn
      WHERE carrid = 'LH'.
      ...
ENDLOOP.
```

Change

```
ls_conn-cityfrom = 'FRANKFURT'.
MODIFY lt_conn FROM ls_conn
      TRANSPORTING cityfrom
      WHERE carrid = 'LH' AND
            connid = '0400'.
```

WHERE only with MODIFY;  
not with MODIFY TABLE

If MODIFY is used with  
WHERE, TRANSPORTING is  
required

Delete

```
DELETE lt_conn WHERE carrid = 'LH'.
```

WHERE only with DELETE;  
not with DELETE TABLE

Figure 115: Mass Access with WHERE Condition



**Hint:** Although key accesses are involved here, the internal table is specified **without the TABLE addition** in these statements.

If you combine MODIFY and WHERE, you must use the TRANSPORTING addition.

When you use WHERE, the runtime needed to locate the addressed line depends primarily on the table type:

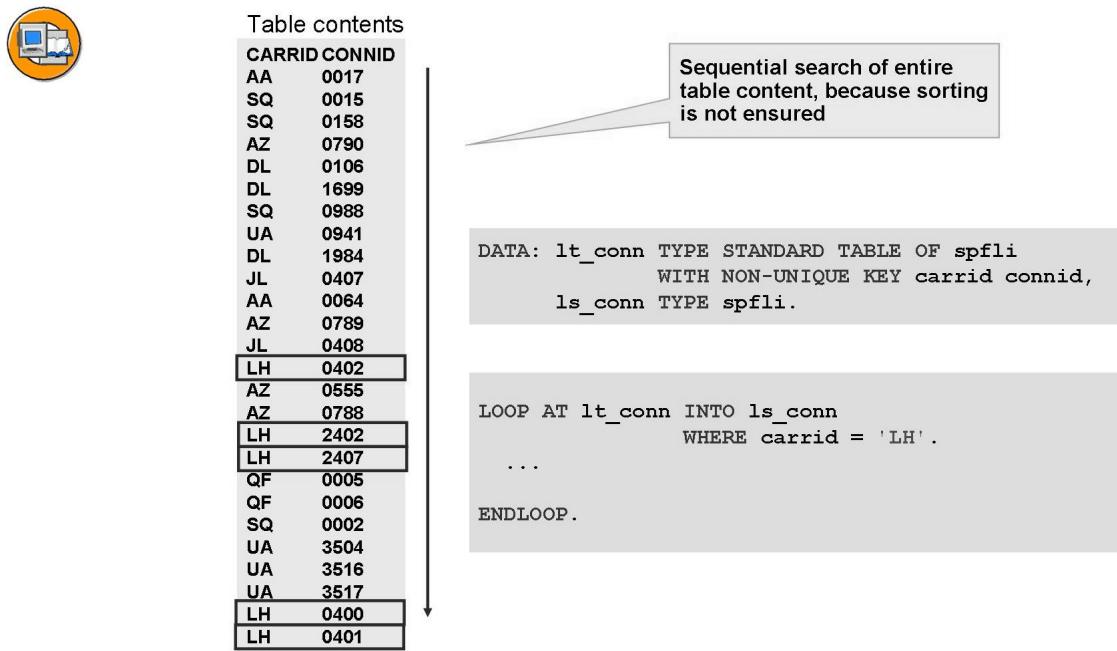


Figure 116: Accessing Standard Tables with WHERE

In **standard tables**, the entire table contents must be searched sequentially to find the desired lines. The runtime requirement increases linearly with the number of lines and can become large.

In **hashed tables**, the hash board is ignored and the table contents are searched completely.



**Hint:** Hashed tables do not provide any better performance than standard tables in mass accesses with WHERE.



## Table contents

CARRID	CONNID
AA	0017
AA	0064
AZ	0555
AZ	0788
AZ	0789
AZ	0790
DL	0106
DL	1699
DL	1984
JL	0407
JL	0408

LH	0400
LH	0401
LH	0402
LH	2402
LH	2407
QF	0005
QF	0006
SQ	0002
SQ	0075
SQ	0158
SQ	0988
UA	0941
UA	3504
UA	3516
UA	3517

```
DATA: lt_conn TYPE SORTED TABLE OF spfli
      WITH NON-UNIQUE KEY carrid connid,
      ls_conn TYPE spfli.
```

```
LOOP AT lt_conn INTO ls_conn
      WHERE carrid = 'LH'.
      ...
ENDLOOP.
```

Binary search for starting point  
(= first row that fulfills the condition).  
Prerequisite: WHERE condition contains  
the first n key fields

Processing ends at first row  
that does not fulfill condition

**Figure 117: Accessing Sorted Tables with WHERE**

In **sorted tables**, the runtime environment can optimize access automatically if the WHERE condition restricts the first n key fields to one value (in other words, when a left-aligned, gap-free part of the key is specified). In this case, the desired entries are not scattered around the table, but instead in a contiguous block (see diagram). In this case, the runtime environment uses a binary search to determine the start of this block and then begins sequential processing from there. As soon as an entry is reached that does not match the key fields, the block is completed and processing ends immediately.



**Hint:** Mass accesses to sorted tables can have much better performance compared to other table types. A prerequisite is that the WHERE condition limits the first n key fields to one value.

## Special Case: Sorted Table with Unique Key

Sorted tables with unique key represent a special application case of this table type. With this case, when you insert a new line, you can immediately find out whether the table already contains a row with an identical key.

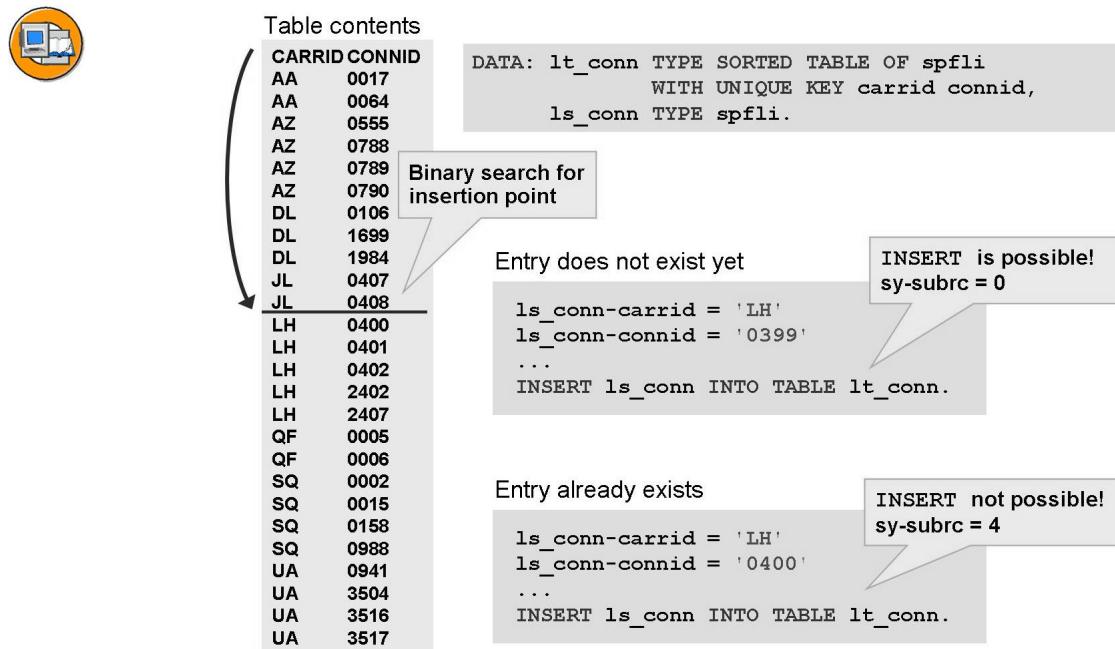


Figure 118: Insert by Key Access into Sorted Table with Unique Key

The new line is only inserted if no line with an identical key is found. In this case, sy-subrc is set to 0. If the insert is not possible, system field sy-subrc is set to 4.



**Hint:** It does not cost any performance to ensure unique keys. After all, when a new line is inserted into a sorted table, the right insert point has to be found for non-unique keys as well.



# Exercise 9: Table Types

## Exercise Objectives

After completing this exercise, you will be able to:

- Select a logical table type for internal tables
- Process data using internal tables

## Business Example

The string from the previous exercise now contains repeated records.

Enhance your program such that the records are held in internal tables in sorted form. Internal tables should only contain unique records. You want to move the duplicate records to a separate internal table.

### Template:

BC402\_IDS\_SPLIT\_ITAB

### Solution:

BC402\_IDS\_TABKIND

### Task 1:

Copy program BC402\_IDS\_SPLIT\_ITAB (or your own program, ZBC402\_##\_SPLIT\_ITAB) to the name **ZBC402\_##\_TABKIND**, where ## is your group number. Make sure the string contains non-unique records.

1. Copy the program and all its subcomponents.
2. Use function module BC402\_CREATE\_SEP\_STRING to generate a string with non-unique records. Set parameter IM\_UNIQUE to constant SPACE.
3. Test the program.

### Task 2:

First buffer the records completely in an internal table and then use a loop to output the contents of the internal table in the list.

1. Define an internal table that can receive flight data sorted by airline, flight number, and flight date (suggested name: **IT\_FLIGHTS**).
2. Insert the completely converted records (the contents of structure GS\_FLIGHT) in this internal table (do not output them).
3. Use a loop to output the contents of the internal table in a list.

*Continued on next page*

4. Test the program and check whether the result actually contains identical records.

### Task 3:

Create a second internal table with the same type. Fill the tables such that the first table only contains unique records, while all additional records are inserted in the second table.

1. Change the definition of the first table so only records with unique table keys can be added.
2. Define another internal table to hold the duplicate records (suggested name: IT\_DOUBLES).
3. Why shouldn't you define this second table with unique keys?

---

---

---

4. Insert each record in one of the two tables. Use the support of the SY-SUBRC return code from INSERT statements to find out whether a record with the same key already exists in the first table.
5. Output the contents of the second internal table at the end of the program.

## Solution 9: Table Types

### Task 1:

Copy program BC402\_IDS\_SPLIT\_ITAB (or your own program, ZBC402\_##\_SPLIT\_ITAB) to the name **ZBC402\_##\_TABKIND**, where ## is your group number. Make sure the string contains non-unique records.

1. Copy the program and all its subcomponents.
  - a) Perform this step in the usual way.
2. Use function module BC402\_CREATE\_SEP\_STRING to generate a string with non-unique records. Set parameter IM\_UNIQUE to constant SPACE.
  - a) See the source code excerpt from the model solution.
3. Test the program.
  - a) Perform this step in the usual way.

### Task 2:

First buffer the records completely in an internal table and then use a loop to output the contents of the internal table in the list.

1. Define an internal table that can receive flight data sorted by airline, flight number, and flight date (suggested name: **IT\_FLIGHTS**).
  - a) See the source code excerpt from the model solution.
2. Insert the completely converted records (the contents of structure GS\_FLIGHT) in this internal table (do not output them).
  - a) See source code excerpt from the model solution
3. Use a loop to output the contents of the internal table in a list.
  - a) See source code excerpt from the model solution
4. Test the program and check whether the result actually contains identical records.
  - a)

*Continued on next page*

## Task 3:

Create a second internal table with the same type. Fill the tables such that the first table only contains unique records, while all additional records are inserted in the second table.

1. Change the definition of the first table so only records with unique table keys can be added.
  - a) See the source code excerpt from the model solution.
2. Define another internal table to hold the duplicate records (suggested name: IT\_DOUBLES).
  - a) See the source code excerpt from the model solution.
3. Why shouldn't you define this second table with unique keys?  
**Answer:** Because some records appear in the string more than twice and you don't want to lose them.
4. Insert each record in one of the two tables. Use the support of the SY-SUBRC return code from INSERT statements to find out whether a record with the same key already exists in the first table.
  - a) See the source code excerpt from the model solution.
5. Output the contents of the second internal table at the end of the program.
  - a) See the source code excerpt from the model solution.

## Result

Source code excerpt from the model solution:

```
REPORT  bc402_ids_tabkind MESSAGE-ID bc402.
```

TYPES:

```
BEGIN OF ty_s_flight_c,
  mandt      TYPE c LENGTH 3,
  carrid     TYPE c LENGTH 3,
  connid     TYPE n LENGTH 4,
  fldate     TYPE n LENGTH 8,
  price      TYPE c LENGTH 20,
  currency   TYPE c LENGTH 5,
  planetype  TYPE c LENGTH 10,
  seatsmax   TYPE n LENGTH 10,
  seatsocc   TYPE n LENGTH 10,
  paymentsum TYPE c LENGTH 22,
  seatsmax_b TYPE n LENGTH 10,
  seatsocc_b TYPE n LENGTH 10,
```

*Continued on next page*

```

        seatsmax_f TYPE n LENGTH 10,
        seatsocc_f TYPE n LENGTH 10,
END OF ty_s_flight_c,

BEGIN OF ty_s_flight,
carrid      TYPE sflight-carrid,
connid      TYPE sflight-connid,
fldate      TYPE sflight-fldate,
price       TYPE sflight-price,
currency    TYPE sflight-currency,
planetype   TYPE sflight-planetype,
seatsmax    TYPE sflight-seatsmax,
seatsocc    TYPE sflight-seatsocc,
END OF ty_s_flight.

CONSTANTS c_number TYPE i VALUE 30.

DATA:
gv_datastring      TYPE string,
gv_set_string      TYPE string,
gs_flight_c        TYPE ty_s_flight_c,
gs_flight          TYPE ty_s_flight.

DATA:
gt_sets           TYPE STANDARD TABLE OF string
                  WITH NON-UNIQUE DEFAULT KEY.

DATA:
gt_flights        TYPE SORTED TABLE OF ty_s_flight
                  WITH UNIQUE KEY carrid connid fldate,
gt_doubles         TYPE SORTED TABLE OF ty_s_flight
                  WITH NON-UNIQUE KEY carrid connid fldate.

START-OF-SELECTION.

* retrieve character string with data (duplicate data sets this time)
CALL FUNCTION 'BC402_CREATE_SEP_STRING'
  EXPORTING
    im_number      = c_number
*      im_table_name = 'SFLIGHT'
*      im_separator = '#'
    im_unique     = ''
  IMPORTING

```

*Continued on next page*

```

        ex_string      = gv_datastring
EXCEPTIONS
        no_data       = 1
        OTHERS        = 2.
IF sy-subrc <> 0.
MESSAGE a038.
ENDIF.

* remove leading and trailing separators
gv_set_string = gv_datastring.
SHIFT gv_set_string RIGHT CIRCULAR BY 2 PLACES.
SHIFT gv_set_string LEFT BY 4 PLACES.

* split into fragments, each corresponding to one data set
SPLIT gv_set_string AT '##' INTO TABLE gt_sets.

LOOP AT gt_sets INTO gv_set_string.

* split into (charlike) fragments corresponding to components
SPLIT gv_set_string AT '#' INTO
        gs_flight_c-mandt
        gs_flight_c-carrid
        gs_flight_c-connid
        gs_flight_c-fldate
        gs_flight_c-price
        gs_flight_c-currency
        gs_flight_c-planetype
        gs_flight_c-seatsmax
        gs_flight_c-seatsocc
        gs_flight_c-paymentsum
        gs_flight_c-seatsmax_b
        gs_flight_c-seatsocc_b
        gs_flight_c-seatsmax_f
        gs_flight_c-seatsocc_f.

* convert fragments into proper data types
MOVE-CORRESPONDING gs_flight_c TO gs_flight.

* store data in tables - one for unique sets, one for doubles
INSERT gs_flight INTO TABLE gt_flights.
IF sy-subrc <> 0.
INSERT gs_flight INTO TABLE gt_doubles.
ENDIF.

ENDLOOP.

```

*Continued on next page*

```
* output result
LOOP AT gt_flights INTO gs_flight.
  WRITE: /
    gs_flight-carrid,
    gs_flight-connid,
    gs_flight-fldate DD/MM/YYYY,
    gs_flight-price CURRENCY gs_flight-currency,
    gs_flight-currency,
    gs_flight-planetype,
    gs_flight-seatsmax,
    gs_flight-seatsocc.
  ENDLOOP.

SKIP.
WRITE: / text-dob COLOR COL_HEADING.

LOOP AT gt_doubles INTO gs_flight.
  WRITE: /
    gs_flight-carrid,
    gs_flight-connid,
    gs_flight-fldate DD/MM/YYYY,
    gs_flight-price CURRENCY gs_flight-currency,
    gs_flight-currency,
    gs_flight-planetype,
    gs_flight-seatsmax,
    gs_flight-seatsocc.
  ENDLOOP.
```



## Lesson Summary

You should now be able to:

- Describe the differences between standard, sorted, and hashed table types
- Tell the difference between key accesses and index accesses to internal tables
- Use the different table types correctly

# Lesson: Special Techniques for Using Internal Tables

## Lesson Overview

In this lesson, you learn about several specific techniques for using internal tables, including the binary search for standard tables, the removal of duplicate lines, and totaling during insertion.



## Lesson Objectives

After completing this lesson, you will be able to:

- Use the **BINARY SEARCH** addition correctly
- Use the **DELETE ADJACENT DUPLICATES** and **COLLECT** statements correctly
- Describe the function of secondary keys for internal tables and apply them correctly.

## Business Example

You want to develop applications in which large volumes of data are stored and processed in internal tables. To ensure efficient data processing, you want to find out about several specific techniques for using internal tables.

## Binary Search for Standard Tables

When sorted tables are accessed using their keys, the runtime system can perform a binary search to reduce the access time for large tables. The **BINARY SEARCH** addition instructs the runtime system to use a binary search during key access to standard tables. This is helpful when the internal table is already sorted by the fields you use to restrict access, but leads to incorrect results when the table is not sorted. The following example illustrates this:

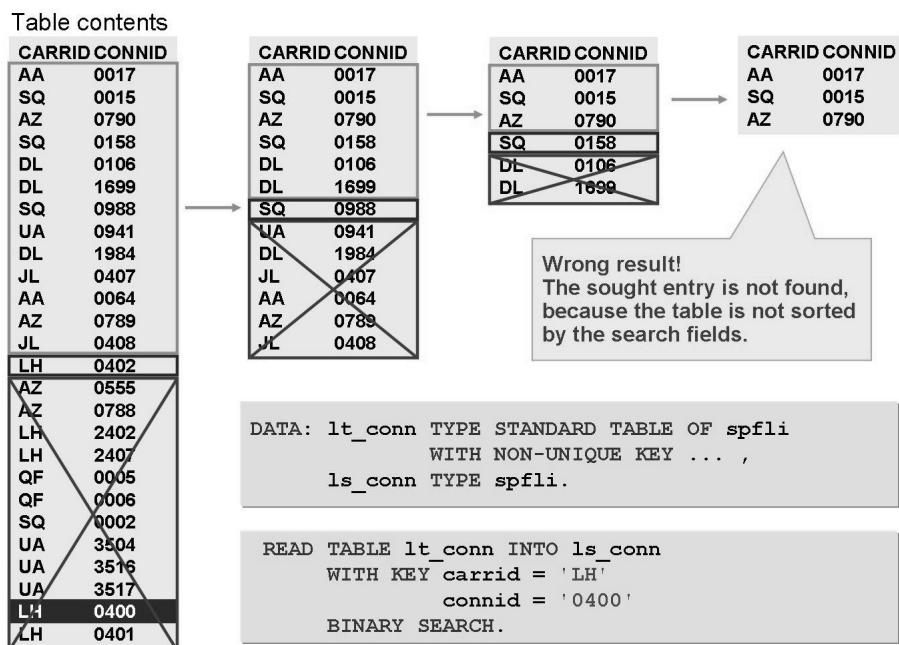


Figure 119: Incorrect Use of the BINARY SEARCH Addition

After the first step, the algorithm continues searching in the first half of the table, because it expects flight LH 0400 before LH 0402. Since the table is not sorted at all, however, the sought entry appears in the second half of the table and is therefore not found.



**Caution:** If you plan to use the BINARY SEARCH addition for key access to standard tables, the table must be sorted by the fields used to restrict access. Otherwise entries may not be found, although they are contained in the table.

Sorting the internal table prior to the binary search is an extra effort that must not be neglected in the performance calculation. The preparation costs only pay back if you do several key accesses with binary search.



**Hint:** You cannot optimize a single access with the BINARY SEARCH addition if you have to sort the table especially for that access.

An explicit binary search can also help you design more efficient LOOPS over standard tables. Once again, a prerequisite is that the table is sorted by the fields that are restricted in the WHERE condition.

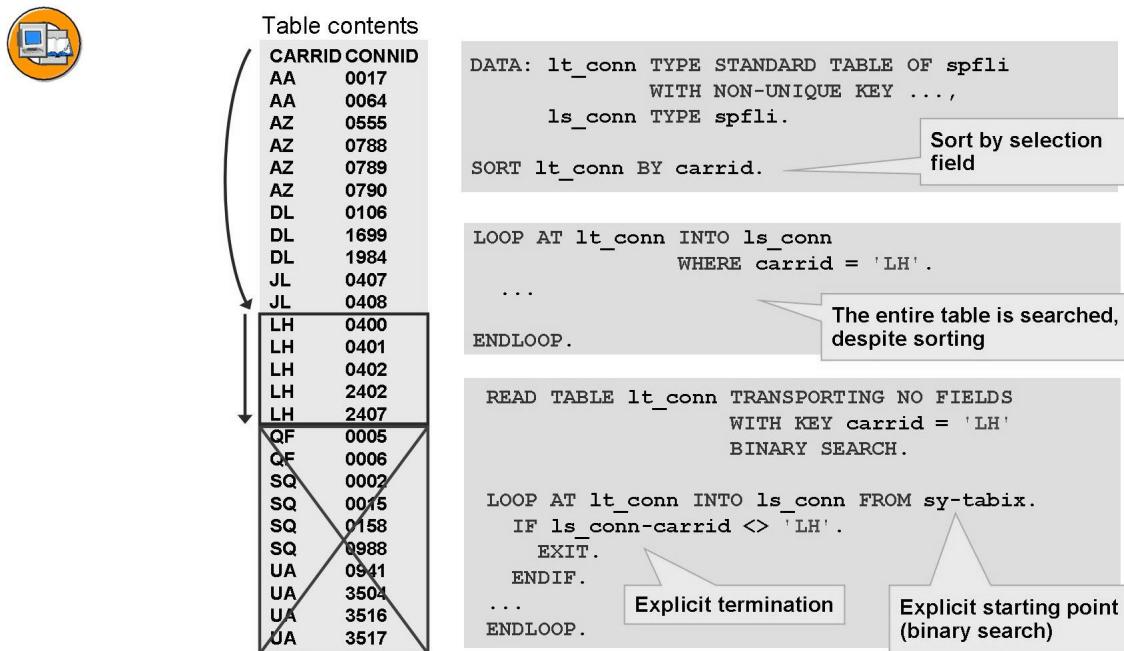


Figure 120: Performance Improved Loop of Standard Table

As a result of the sort sequence, the entries that satisfy the WHERE condition are located in a contiguous block. The READ TABLE ... BINARY SEARCH statement determines the position of the first item in this block (contents of system field sy-tabix). Since only the position is needed at first, this line does not have to be read (TRANSPORTING NO FIELDS addition).



**Hint:** Note that READ TABLE always returns the first entry if several entries satisfy the condition.

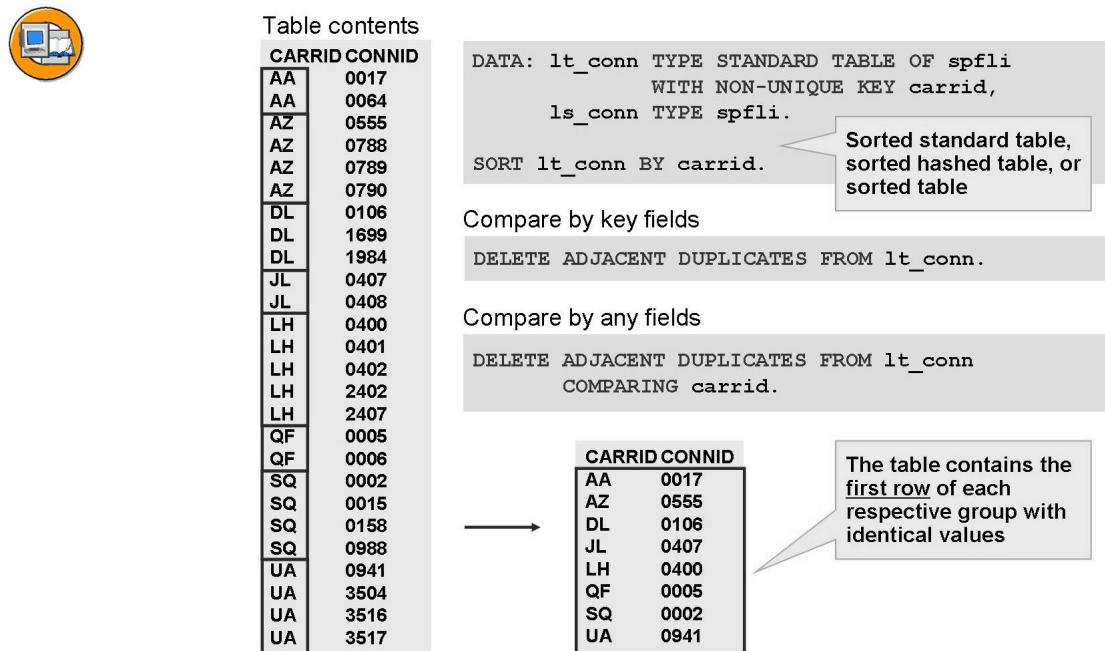
During the subsequent LOOP, the determined line is used as the starting point (FROM addition). An explicit termination (EXIT statement) ends processing as soon as a line is reached that does not satisfy the condition (that is, lies outside the block).



**Note:** The illustrated source code implements the same algorithm that the runtime system uses automatically for sorted tables.

## Deleting Duplicates (DELETE ADJACENT DUPLICATES)

The DELETE ADJACENT DUPLICATES statement lets you remove duplicate lines from an internal table. All lines whose contents of certain fields are identical to the contents of the immediately preceding lines are deleted.



**Figure 121: The DELETE ADJACENT DUPLICATES Statement**

In the example, the internal table contains several instances of multiple lines with the same value in the CARRID column. The DELETE ADJACENT DUPLICATES statement leaves the first line of each group in the internal table and deletes all the others.



**Caution:** The comparison only compares **adjacent** table lines. For the statement to find and remove all duplicates, you have to make sure that the table is **sorted** by the comparison fields.

The COMPARING edition lets you control which fields are compared. The following options are available:

#### Without the COMPARING addition

The contents of the key fields are compared. This variant is only sensible for internal tables with nonunique keys, since unique keys mean there cannot be two lines with the same contents in the key fields in the first place. The table must be sorted by the table key.

#### COMPARING col1 col2 ...

The contents of columns col1, col2, ... are compared. You can specify any number of column names, as well as any combination of key fields and non key fields. For sorted tables, this variant is usually only sensible when you specify the first n key fields.

## COMPARING ALL FIELDS

All fields are compared. Therefore, only lines that are completely identical to their preceding lines are deleted.

## Total during Insert (COLLECT)

The COLLECT statement is a special technique for filling an internal table. The table is aggregated during the insert operation.

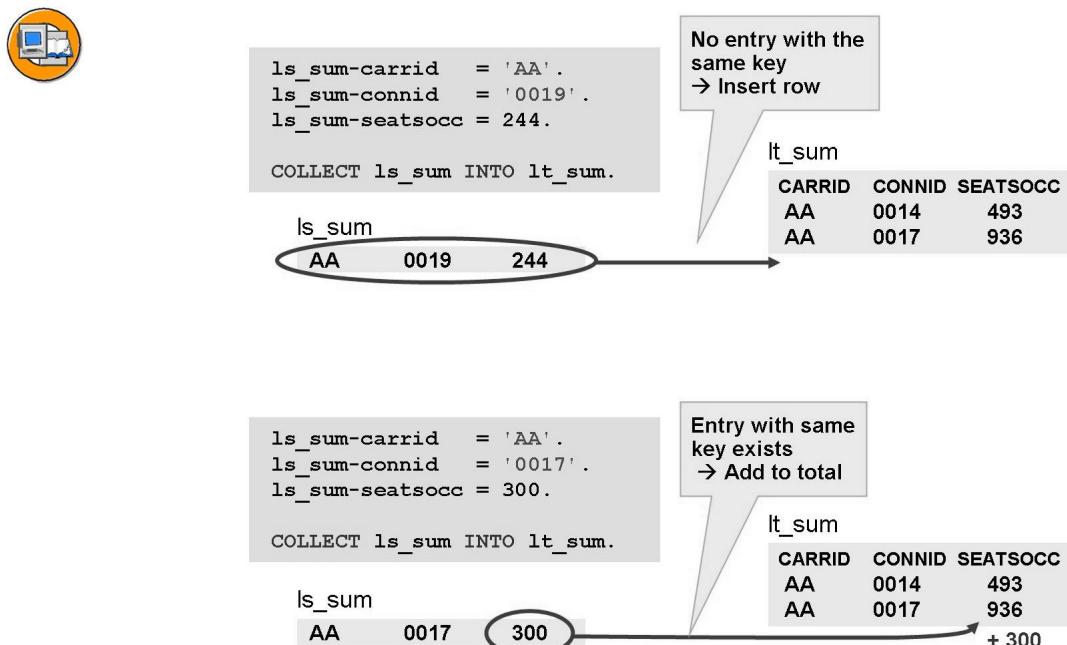


Figure 122: COLLECT - Insert or Add to Total

If the table does not have a line with the same key yet, the COLLECT statement has the same effect as inserting by key access.

If the table already contains one or more lines with this key, the values of all non key fields in the new line are added to the corresponding values in the top line.



**Hint:** The COLLECT statement is only allowed when all non key fields are numeric (types i, p, or f). The **syntax check** reports an error if this prerequisite is not met.

The easiest way to fulfill the condition is to define an internal table with **standard key** (WITH DEFAULT KEY addition). The standard key is defined such that all non-numeric fields are key fields and all non-key fields have a numeric type.

The following example shows how the COLLECT statement can be used to aggregate an existing internal table - that is, form subtotals by group.



```

LOOP AT lt_flight INTO ls_flight.
  MOVE-CORRESPONDING ls_flight TO ls_sum.
  COLLECT ls_sum INTO lt_sum.
ENDLOOP.

```

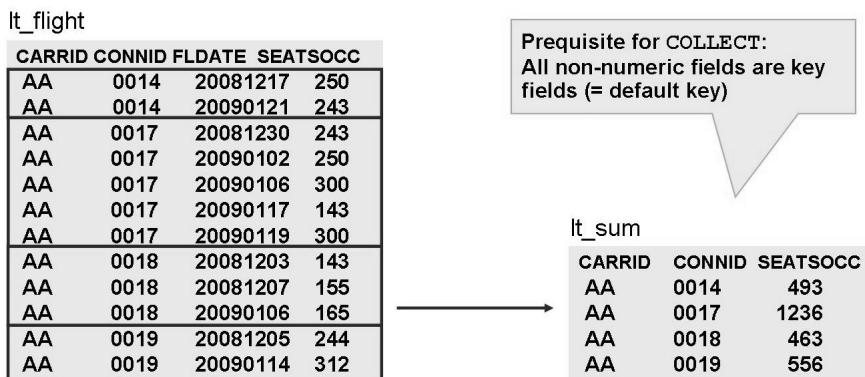


Figure 123: Aggregating an Internal Table with COLLECT

The COLLECT statement is a key access, which means it can be used for all three table types.



**Caution:** We recommend using the COLLECT statement **only for sorted or hashed tables**. The statement can have poor performance with standard tables, because the table to be filled is first searched sequentially during each execution of COLLECT. This is particularly true as the table grows over time.

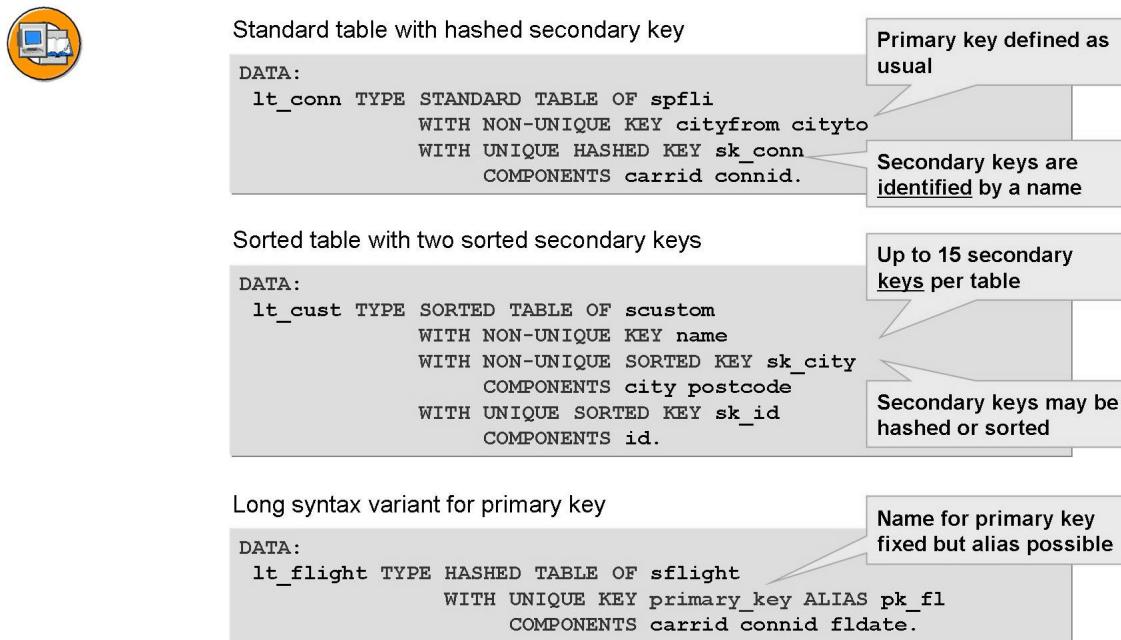
## Secondary Keys for Internal Tables (New in SAP NetWeaver 7.0 EhP2)

As of SAP NetWeaver 7.0 EhP2 it is possible to define secondary keys for internal tables. Secondary keys can be hashed or sorted. A sorted secondary key may be either unique or nonunique. The maximum number of secondary keys per internal table is 15.

### Defining Secondary Keys

You can define secondary keys for global table types in the TYPES statement or the DATA statement. For global table types in *ABAP Dictionary* a new tab *Secondary Key* is available.

The following figure gives examples for the definition of secondary keys:



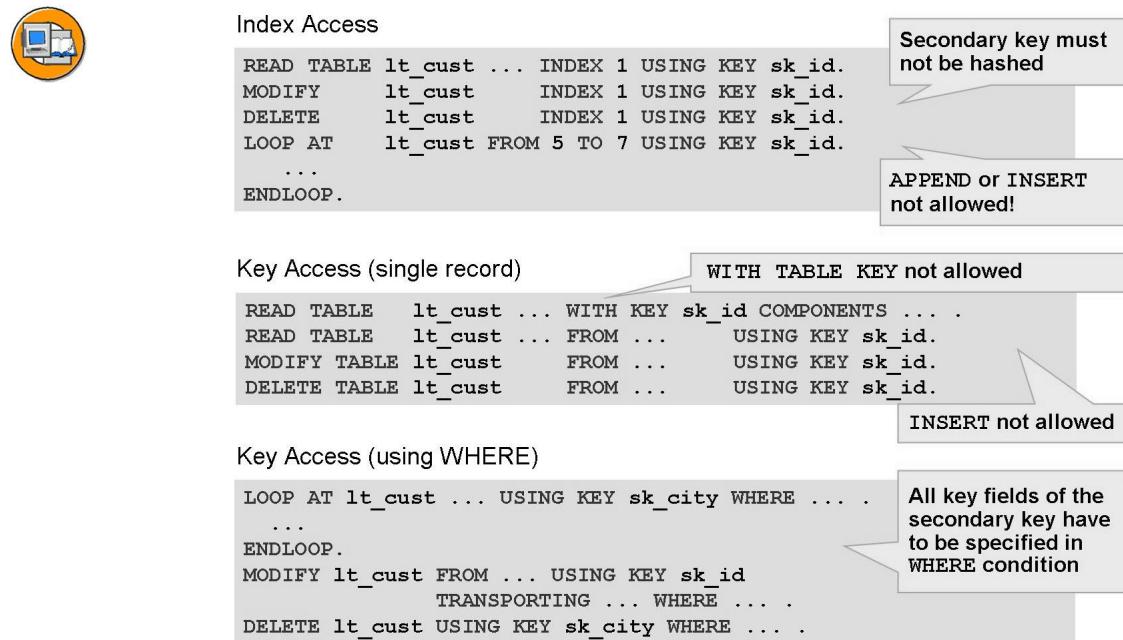
**Figure 124: Examples for Internal Tables With Secondary Keys**

For each secondary key a key name has to be specified. The name is arbitrary but has to be unique for this table.

Along with the introduction of secondary keys the syntax for defining the primary key has been enhanced (see above). For compatibility reasons, the additions are all optional.

### Using Secondary Keys

A secondary key is used in a processing statement by specifying the key name with additions USING KEY key\_name or WITH KEY key\_name. This is shown in the next figure:



**Figure 125: Using secondary keys in operations**

As can be seen above, secondary keys may be used in most processing statements for internal tables. They are not allowed in:

- APPEND
- INSERT (either key or index access)
- READ TABLE ... WITH **TABLE KEY** ...
- DELETE TABLE ... WITH TABLE KEY ...

No automatic selection of a secondary key takes place. If no secondary key is specified during a processing statement, the primary key or primary table index is always used. There is a warning from the syntax check if a suitable secondary key is defined but not used in a given processing statement.



**Hint:** When using a sorted secondary key in an index access, the specified index will be applied according to the sort order in the secondary key.

After an access via a sorted secondary key the SY-TABIX field will contain the position of the line in the secondary key.

In a key access (single record or with WHERE) the components in the condition have to match the components of the secondary key.

## Update of Secondary Keys

To reduce the costs for secondary keys, they are not always updated immediately when the content of the internal table is changed. Unique and nonunique secondary keys are treated differently.

A **nonunique** secondary key is only created when it is used explicitly for the first time. After any kind of write access to the internal table, the update of the secondary key is delayed until the next statement that explicitly uses this secondary key (lazy update). This is shown in the following example:

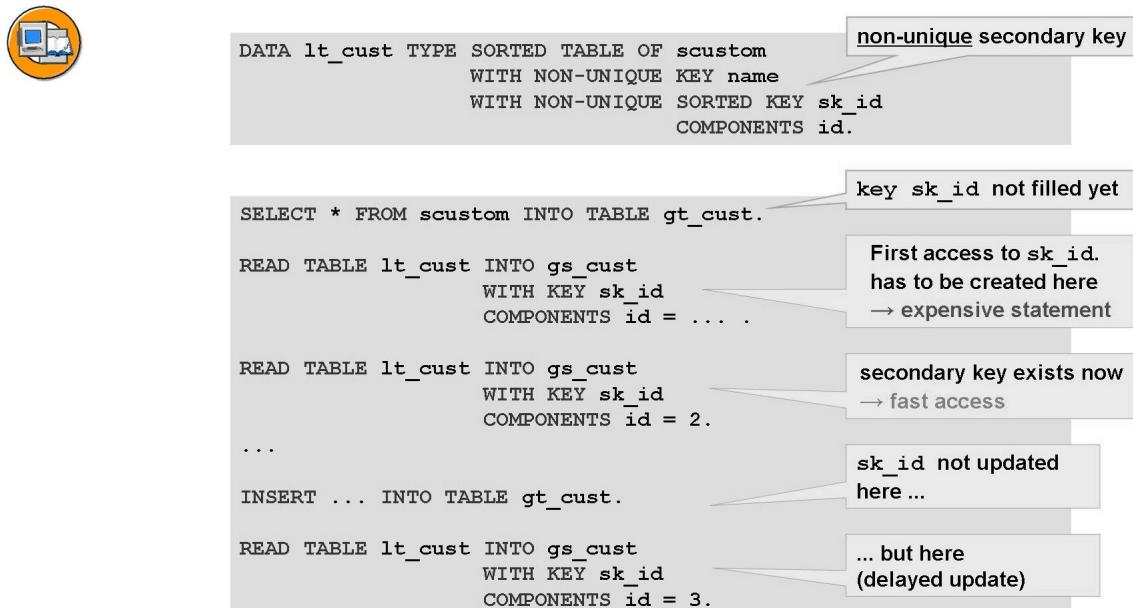
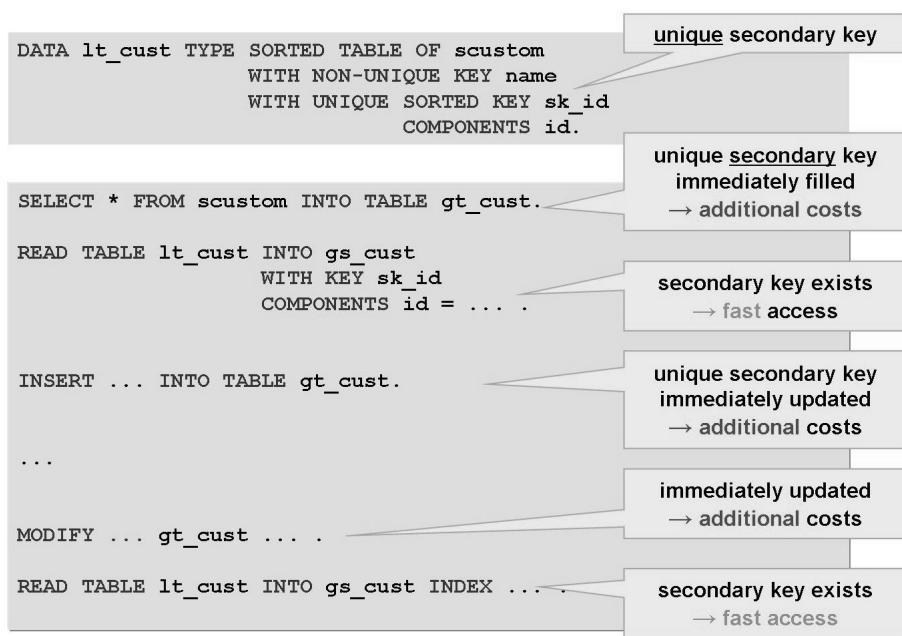


Figure 126: Lazy update of nonunique secondary key



**Hint:** As a consequence of the lazy update, the first access through a secondary key will appear very expensive in the performance analysis tools.

A **unique** secondary key forms a boundary condition for the internal table and is therefore treated differently:



**Figure 127: Update of Unique Secondary Key is Not Delayed**

For any write access that adds or removes lines, the update of the secondary key is done immediately. If the uniqueness is violated a catchable exception is raised (Exception class CX\_SY\_ITAB\_DUPLICATE\_KEY).

For MODIFY statements that change the key fields of a unique secondary key the secondary key is also updated immediately. If the uniqueness is violated a **non catchable** exception is raised.

Only for direct changes through field symbols or data references (see next lesson) the update of the unique secondary key is delayed. But different to nonunique keys, the unique secondary key is updated at the very next access to the internal table. It is not necessary that this secondary key is actually used in this access.

## Recommendations

When using secondary keys the following should be kept in mind:

### Only use secondary key when used often

Do not neglect the additional costs for creating/updating the secondary key.  
Only use secondary keys if they are used often enough.



**Hint:** Building a secondary key is even more expensive than sorting a standard table.

### Prefer nonunique secondary keys

Prefer nonunique secondary keys because the update is always delayed and only done if the respective secondary key is actually used. In addition, when an internal table has a unique secondary key any write access to the table bears the risk of runtime errors due to a violation of the uniqueness.

### Be careful with index operations

After an operation that uses a sorted secondary key, the SY-TABIX reflects the position of the respective line in this secondary key. If you want to use SY-TABIX to address the line again in a subsequent index access, you have to make sure that you specify the same secondary key.



#### Secondary key only when used often

- Building secondary key is expensive
- Only pays back when used often

#### Prefer non-unique secondary keys

- update always delayed
- update only when secondary key is used
- no danger of runtime errors

#### Be careful with index operations

- Sort order different in primary and secondary key
- SY-TABIX reflects position in secondary key
- INDEX access according to specified key

Figure 128: Recommendations for using secondary keys



# Exercise 10: Using Special Techniques for Internal Tables

## Exercise Objectives

After completing this exercise, you will be able to:

- Use the DELETE ADJACENT DUPLICATES statement correctly
- Use the COLLECT statement correctly

## Business Example

You want to enhance a program that outputs a customer's flight bookings. Based on the list of bookings, you also want to output the total bookings by currency and an alphabetical list of the involved travel agencies.

### Template:

BC402\_IDT\_ITAB\_TECH

### Solution:

BC402\_IDS\_ITAB\_TECH

### Task 1:

Copy the template BC402\_IDT\_ITAB\_TECH to the name **ZBC402\_##\_ITAB\_TECH**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
2. Which input options are available on the selection screen?

---

---

---

3. Which data is output in the list? Which data do you need, but is still missing?

---

---

---

*Continued on next page*

## Task 2:

Implement subroutine GET\_TRAVELAGS to determine the travel agencies that made the bookings. Make sure of the following:

Each travel agency only appears once

The travel agencies are sorted alphabetically by name



**Hint:** There is no need to read from the database or access global data objects of the program from within the subroutine. All the necessary data is provided in the USING parameter of the subroutine.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which table type does CHANGING parameter ct\_travelags have?

---

---

---

2. Which components does the line type of ct\_travelags have?

---

---

---

3. Implement a loop of the bookings and fill ct\_travelags with the data for **all** travel agencies initially.
4. Now remove all duplicate entries from ct\_travelags.
5. Now sort ct\_travelags by the travel agency name.

*Continued on next page*

### Task 3:

Implement subroutine GET\_SUMS to calculate the booking totals separated by currency.



**Hint:** There is no need to read from the database or access global data objects of the program from within the subroutine. All the necessary data is provided in the USING parameter of the subroutine.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which table type does CHANGING parameter ct\_sums have?

---

---

---

2. Which component does the line type of ct\_sums contain?

---

---

---

3. Implement a loop of the bookings and fill ct\_sums. Use a technique that forms the totals at the same time as it fills the data.

4. Why can the technique for calculating totals during filling be used here?

---

---

---

## Solution 10: Using Special Techniques for Internal Tables

### Task 1:

Copy the template BC402\_IDT\_ITAB\_TECH to the name **ZBC402\_##\_ITAB\_TECH**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Which input options are available on the selection screen?  
**Answer:** An airline customer number
3. Which data is output in the list? Which data do you need, but is still missing?

**Answer:** Data output:

- Customer's name and form of address
- A list of all the customer's bookings (without cancellations)

Still missing:

- A list of total bookings (separated by currency)
- A list of travel agencies who made the bookings

### Task 2:

Implement subroutine GET\_TRAVELAGS to determine the travel agencies that made the bookings. Make sure of the following:

Each travel agency only appears once

The travel agencies are sorted alphabetically by name



**Hint:** There is no need to read from the database or access global data objects of the program from within the subroutine. All the necessary data is provided in the USING parameter of the subroutine.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which table type does CHANGING parameter ct\_travelags have?

**Answer:** Standard table with non-unique key.

*Continued on next page*

2. Which components does the line type of ct\_travelags have?

**Answer:**

**AGENCYNUM:** Travel agency number

**NAME:** Travel agency name

**CITY:** City

3. Implement a loop of the bookings and fill ct\_travelags with the data for **all** travel agencies initially.
  - a) See the source code excerpt from the model solution.
4. Now remove all duplicate entries from ct\_travelags.
  - a) See the source code excerpt from the model solution.
5. Now sort ct\_travelags by the travel agency name.
  - a) See the source code excerpt from the model solution.

### Task 3:

Implement subroutine GET\_SUMS to calculate the booking totals separated by currency.



**Hint:** There is no need to read from the database or access global data objects of the program from within the subroutine. All the necessary data is provided in the USING parameter of the subroutine.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which table type does CHANGING parameter ct\_sums have?

**Answer:** Sorted table with unique key.

2. Which component does the line type of ct\_sums contain?

**Answer:**

**FORCURRAM:** Price of booking in foreign currency (dependent on booking location)

**FORCURKEY:** Payment currency

3. Implement a loop of the bookings and fill ct\_sums. Use a technique that forms the totals at the same time as it fills the data.

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

4. Why can the technique for calculating totals during filling be used here?

**Answer:** Because all non-key fields (FORCURRAM here) in the table are numeric.

*Continued on next page*

## Result

Source code excerpt from the model solution:

```

REPORT  bc402_ids_itab_tech.

TYPES: BEGIN OF gty_s_sums,
        forcuram  TYPE bc402_scus_book-forcuram,
        forcurkey TYPE bc402_scus_book-forcurkey,
    END OF gty_s_sums.

TYPES: BEGIN OF gty_s_travelags,
        agencynum  TYPE bc402_scus_book-agencynum,
        name       TYPE bc402_scus_book-name,
        city       TYPE bc402_scus_book-city,
    END OF gty_s_travelags.

TYPES:
      gty_t_sums          TYPE SORTED TABLE OF gty_s_sums
                           WITH UNIQUE KEY forcurkey,

      gty_t_bookings      TYPE STANDARD TABLE OF bc402_scus_book
                           WITH NON-UNIQUE KEY
                           carrid connid fldate bookid,

      gty_t_travelags    TYPE STANDARD TABLE OF gty_s_travelags
                           WITH NON-UNIQUE KEY agencynum.

DATA:
      gs_customer  TYPE scustom,
      gt_bookings   TYPE gty_t_bookings,
      gt_sums       TYPE gty_t_sums,
      gt_travelags  TYPE gty_t_travelags.

PARAMETERS:
      pa_cust  TYPE sbook-customid DEFAULT '00000001'.

START-OF-SELECTION.

      SELECT SINGLE * FROM scustom INTO gs_customer
        WHERE id = pa_cust.

      SELECT * FROM bc402_scus_book
        INTO TABLE gt_bookings
        WHERE customid = pa_cust

```

*Continued on next page*

```

        AND cancelled <> 'X'.

        PERFORM get_sums USING      gt_bookings
                           CHANGING gt_sums.

        PERFORM get_travelags USING      gt_bookings
                           CHANGING gt_travelags.

        PERFORM output_list USING gs_customer
                           gt_bookings
                           gt_sums
                           gt_travelags.

*&-----*
*&      Form  GET_SUMS
*&-----*
FORM get_sums USING      pt_bookings TYPE gty_t_bookings
                  CHANGING ct_sums      TYPE gty_t_sums.

DATA:
ls_bookings LIKE LINE OF pt_bookings,
ls_sums      LIKE LINE OF ct_sums.

LOOP AT pt_bookings INTO ls_bookings.

MOVE-CORRESPONDING ls_bookings TO ls_sums.
COLLECT ls_sums INTO ct_sums.

ENDLOOP.

ENDFORM.          " GET_SUMS

*&-----*
*&      Form  GET_TRAVELAGS
*&-----*
FORM get_travelags USING      pt_bookings  TYPE gty_t_bookings
                           CHANGING ct_travelags TYPE gty_t_travelags.

DATA:
ls_bookings  LIKE LINE OF pt_bookings,
ls_travelags LIKE LINE OF ct_travelags.

LOOP AT pt_bookings INTO ls_bookings.

```

*Continued on next page*

```

MOVE-CORRESPONDING ls_bookings TO ls_travelags.
APPEND ls_travelags TO ct_travelags.

ENDLOOP.

SORT ct_travelags BY agencynum.

DELETE ADJACENT DUPLICATES FROM ct_travelags
      COMPARING agencynum.

SORT ct_travelags BY name.

ENDFORM.          " GET_TRAVELAGS

*&-----*
*&      Form  OUTPUT_LIST
*&-----*

FORM output_list USING ps_customer  TYPE scustom
                  pt_bookings  TYPE gty_t_bookings
                  pt_sums      TYPE gty_t_sums
                  pt_travelags TYPE gty_t_travelags.

DATA:
ls_bookings  LIKE LINE OF pt_bookings,
ls_sums      LIKE LINE OF pt_sums,
ls_travelags LIKE LINE OF pt_travelags.

DATA lv_text TYPE string.

CONCATENATE text-wcm
              ps_customer-form
              ps_customer-name
              INTO lv_text
              SEPARATED BY space.
CONDENSE lv_text.

WRITE: / lv_text.
SKIP.

WRITE / text-lob.
ULINE.
SKIP.

LOOP AT pt_bookings INTO ls_bookings.
WRITE: /

```

*Continued on next page*

```
        ls_bookings-bookid,
        ls_bookings-carrid RIGHT-JUSTIFIED,
        ls_bookings-connid ,
        ls_bookings-fldate,
        ls_bookings-cityfrom RIGHT-JUSTIFIED,
        '->',
        ls_bookings-cityto,
        ls_bookings-forcuram CURRENCY ls_bookings-forcurkey,
        ls_bookings-forcurkey.
ENDLOOP.
SKIP 2.

WRITE: / text-sum.
ULINE.
SKIP.

LOOP AT pt_sums INTO ls_sums.
  WRITE: /
    ls_sums-forcuram CURRENCY ls_sums-forcurkey,
    ls_sums-forcurkey.
ENDLOOP.
SKIP 2.

WRITE / text-tag.
ULINE.
SKIP.

LOOP AT pt_travelags INTO ls_travelags.
  WRITE: /
    ls_travelags-name,
    ls_travelags-city.
ENDLOOP.

ENDFORM.          " OUTPUT_LIST
```



## Lesson Summary

You should now be able to:

- Use the BINARY SEARCH addition correctly
- Use the DELETE ADJACENT DUPLICATES and COLLECT statements correctly
- Describe the function of secondary keys for internal tables and apply them correctly.

# Lesson: Using Data References and Field Symbols

## Lesson Overview

In this unit, you learn about data references and field symbols, two different concepts for pointer variables in ABAP. Combined with generic data types, these concepts play a major part in dynamic programming. In this lesson, however, we concentrate on fully typed data references and field symbols.

You first learn what data references and field symbols are, the differences between them, and how to use them. In the last section, you learn how you can use data references and field symbols to improve performance – vastly in some cases – when accessing internal tables.



## Lesson Objectives

After completing this lesson, you will be able to:

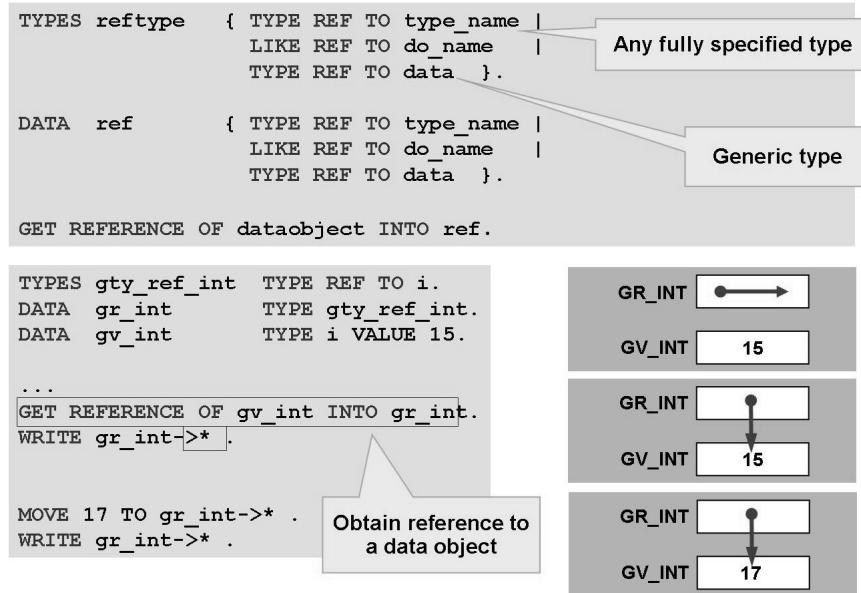
- Explain the syntax for working with data references and field symbols
- Use field symbols and data references to access the content of internal tables

## Business Example

You want to improve performance when accessing the contents of internal tables. You have heard that data references and field symbols make this possible. You therefore want to learn about the syntax for using data references and field symbols, as well as how to use data references and field symbols to improve performance when accessing internal tables.

## Data References

Data references and object references were introduced alongside field symbols as part of the enhancements in *SAP R/3 4.6A*. Since this point, ABAP features full “reference semantics”.



**Figure 129: Data References**

Data reference variables contain data references, or pointers to data objects. You can use the TYPE REF TO addition for the TYPES statement, to define a reference type for a data object. You can specify the data type explicitly or choose the generic variation using TYPE REF TO DATA. In this case, your data reference can point to any type of data object.

The corresponding DATA statement defines the data reference variable itself. Reference variables are data objects that can contain the address of any other data object of the specified type.

Data references involve using reference semantics, that is, when a data reference variable is accessed, the data reference itself is addressed, which means any changes affect **the addresses**.

Data reference variables are handled in ABAP like other data objects with an elementary data type. This means a reference variable can be defined not only as a single field, but also as the smallest indivisible unit of complex data objects such as structures or internal tables.

After it has been defined, a data reference variable is initial, and thus contains a blank pointer. For a data reference variable to contain a reference that points to a data object, you must use the statement GET REFERENCE OF to obtain a reference to a data object that has already been defined.

It can also be assigned an existing data reference from another data reference variable, or a data object can be created dynamically with the reference. (More about this later.)

Statically typed data references can be dereferenced directly using the dereferencing operator `->*`. This means they directly access the content of the data object that points to the reference. For compatibility reasons, you must dereference generically typed data references (TYPE REF TO data) and assign them to a field symbol, which you can then use to access the content. (This method is demonstrated again later.)

When data references are typed statically with a structure type, you can address the components of the referenced data object directly, using the component selector `->`, and use them in any operand item.

### Example:

```
DATA: gs_flight      sflight,
      gr_flight TYPE REF TO sflight.

GET REFERENCE OF gs_flight INTO gr_flight.

gr_flight->flddate = gr_flight->flddate + 5.

WRITE: / gr_flight->seatsmax.
```

The component selector therefore corresponds here to the hyphen for regular component access to structured data objects.



### Validity of References – Logical Expression

`... ref IS [NOT] BOUND ...`

The expression `ref IS [NOT] BOUND` is used to query whether the reference variable `ref` contains a valid reference. `ref` must be a data or object reference variable.

When a data reference is involved, this logical expression is true if it can be dereferenced. When an object reference is involved, it is true if it points to an object. The logical expression is always false if `ref` contains a null reference.

In contrast, you can only use the expression `... ref IS [NOT] INITIAL ...` to determine whether `ref` contains the null reference or not.

### Field Symbols

Dereferenced pointers as field symbols were available in ABAP even before the introduction of data references. Field symbols allow you “symbolic” access to an existing data object. All accesses that you make to the field symbol are made to the data object assigned to it. Therefore, you can access only the **content** of the data object to which the field symbol points. This technique is referred to as the “value semantics of field symbols”.



```
FIELD-SYMBOLS <fs> TYPE|LIKE ... | TYPE ANY }.
ASSIGN dataobject TO <fs>.
UNASSIGN <fs>.
... <fs> IS ASSIGNED ...
```

Generic or complete type definition

```
DATA gv_int TYPE i VALUE 15.
FIELD-SYMBOLS <fs_int> TYPE i.
```

<FS\_INT> → 15 INT

```
ASSIGN gv_int TO <fs_int>.
WRITE: / gv_int, <fs_int>.
```

<FS\_INT> → 15 GV\_INT

```
<fs_int> = 17.
WRITE: / gv_int, <fs_int>.
```

<FS\_INT> → 17 GV\_INT

```
UNASSIGN <fs_int>.
IF <fs_int> IS ASSIGNED.
  WRITE: / gv_int, <fs_int>.
ELSE.
  WRITE: / 'field symbol not assigned' (fna).
ENDIF.
```

<FS\_INT> → 17 GV\_INT

**Figure 130: Field Symbols**

You declare field symbols using the FIELD-SYMBOLS statement. Note that the angle brackets (<>) in the field symbol name are part of the syntax.

You use the ASSIGN statement to assign a data object to the field symbol.

By specifying a type for the field symbol, you can ensure that only compatible objects are assigned to it.

### Example:

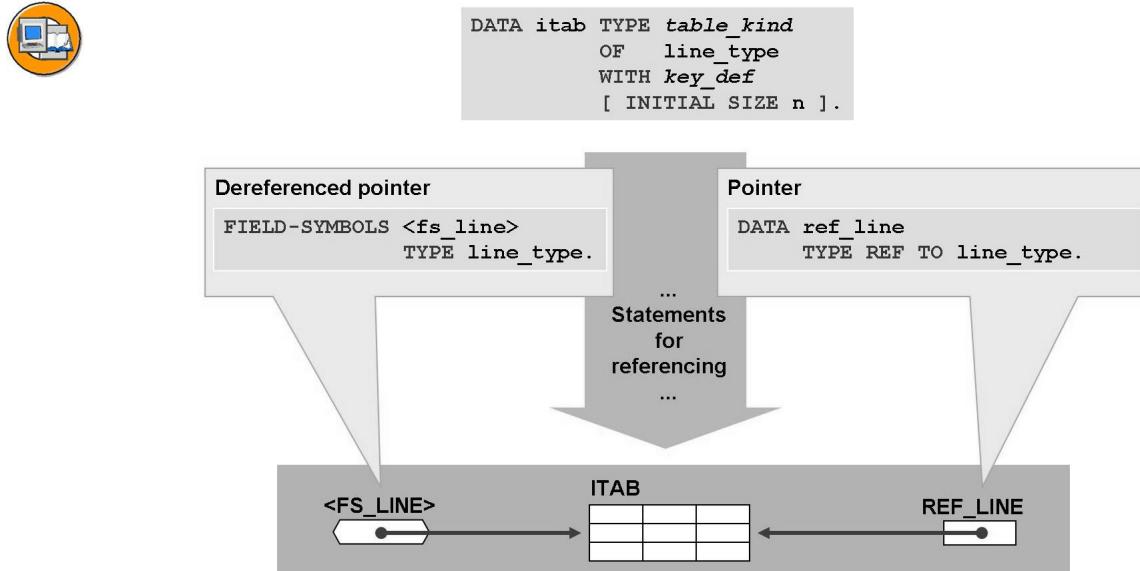
```
DATA: gv_date TYPE d VALUE '20040101',
      gv_time TYPE t.
FIELD-SYMBOLS: <fs_date> TYPE d,
               <fs_time> TYPE t.
ASSIGN: gv_date TO <fs_date>,
        gv_time TO <fs_time>.

* possible?
<fs_time> = <fs_date>.
```

Use the expression IS ASSIGNED to find out whether the field symbol is assigned to a field. The statement UNASSIGN sets the field symbol so that it points to nothing. The logical expression IS ASSIGNED is then false.

## Accessing Internal Tables using Data References and Field Symbols

When combined with internal tables, they open interesting new access options, which we examine below.



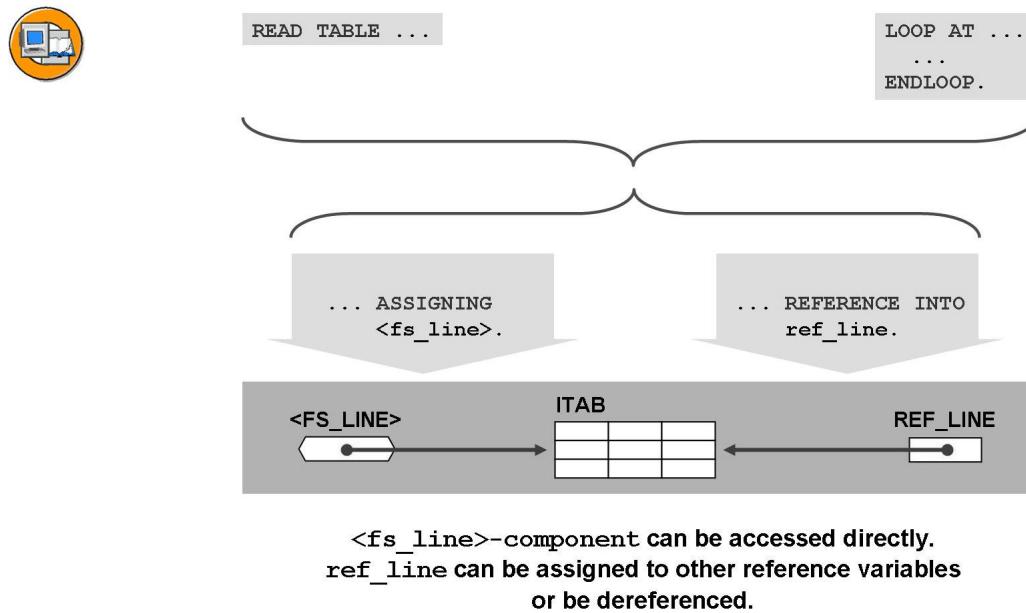
**Figure 131: Using Field Symbols and References for Line Access**

You can use one of the statements described above to “point” both field symbols and references to individual lines in an internal table. This makes it possible to use these field symbols or the dereferencing operator to access these records. Alternatively, you can also use references to the records directly – for example, copy them to other reference variables.



**Hint:** Field symbols and references continue to point to the assigned row even after the internal table is resorted!

Unless there are compelling reasons to the contrary, you should always type the field symbols and references using types that are compatible with the internal table. Generic types are also possible within the dynamic programming framework. In this case, the CASTING addition **cannot** be used.



**Figure 132: Referencing during Read and Loop Processing**

When you read a table row using READ TABLE or a series of table rows using LOOP AT, you can assign the rows of the internal table to a field symbol using the ASSIGNING <field\_symbol> addition. The <field\_symbol> field symbol then points to the line that you assigned, allowing you to access the field content directly. Consequently, the system does not have to copy the entry from the internal table to the work area and back again.

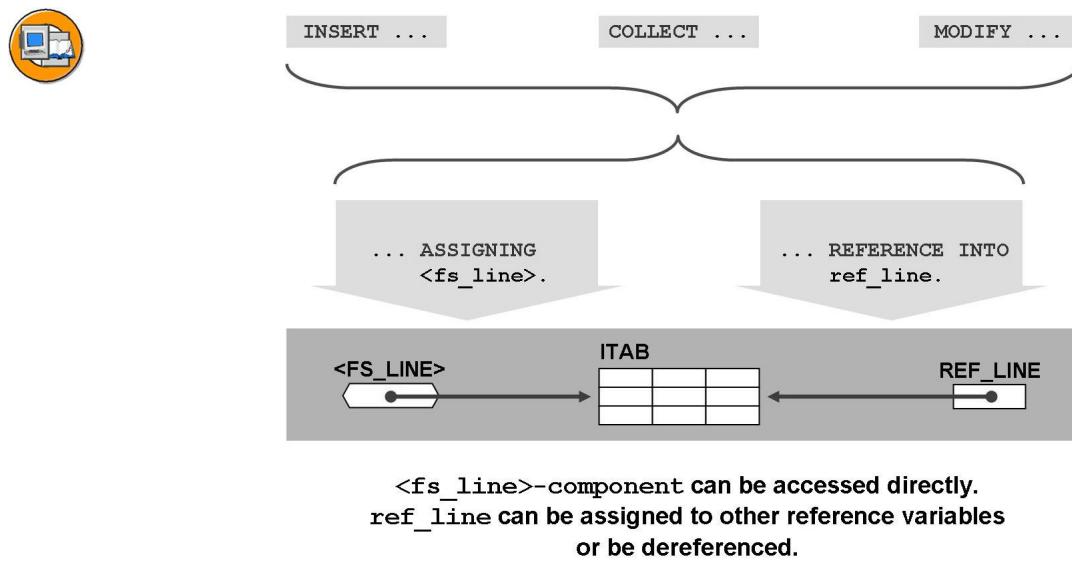
However, there are also certain restrictions when you use this technique:

- You can only change the contents of key fields if the table is a standard table.
- You cannot reassign field symbols within the loop. The statements ASSIGN do TO <fs\_line> and UNASSIGN <fs\_line> cause runtime errors.
- The SUM statement is **not** available for control level processing.

The same restrictions apply for the variant REFERENCE INTO ref\_name, which you can use to “point” a reference to the lines.

This technique is particularly useful for accessing multiple table lines or nested tables within a loop. Copying values to and from the work area in this kind of operation is particularly runtime-intensive.

→ **Note:** These access techniques are also useful for dynamic programming.



**Figure 133: References to Inserted or Changed Single Lines**

If you use one of the above three statement to exchange data with the internal table through a work area, you can also use the optional additions **ASSIGNING <field\_symbol>** and **REFERENCE INTO ref\_name** to “point” to a field symbol or reference to the processed line.



# Exercise 11: Field Symbols and Data References with Internal Tables

## Exercise Objectives

After completing this exercise, you will be able to:

- Use field symbols when working with internal tables
- Use data references when working with internal tables

## Business Example

You have been asked to improve a program that outputs a list of bookings. To improve performance, you will replace the work areas with field symbols and data references.

### Template:

BC402\_IDS\_ITAB\_TECH

### Solution:

BC402\_IDS\_FSYM\_DREF

### Task 1:

Copy program BC402\_IDS\_ITAB\_TECH (or your own program, ZBC402\_##\_ITAB\_TECH) to the name **ZBC402\_##\_FSYM\_DREF**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
2. Activate and test the program.

### Task 2:

Only edit subroutine get\_sums at first. Where it makes sense, use **data references** instead of explicit work areas.

1. Examine the source code of the subroutine. For which of the two internal tables can the work area be replaced completely by a data reference?

---

---

---

2. Define a suitably typed data reference.

*Continued on next page*

3. Modify the LOOP statement such that the data reference is filled during each loop pass, and not the work area.
4. Replace structure ls\_bookings with the data reference and then deactivate (flag as comments) the definition of structure ls\_bookings.

### Task 3:

Now edit subroutine get\_travelags. Where it makes sense, use **field symbols** instead of explicit work areas.

1. Examine the source code of the subroutine. For which of the two internal tables can the work area be replaced completely by a field symbol?

---

---

---

2. Define a suitably typed field symbol.
3. Modify the LOOP statement such that the current line is assigned to the field symbol during each loop pass, and not copied to the work area.
4. Replace structure ls\_bookings with the field symbol and deactivate (flag as comments) the definition of structure ls\_bookings.

### Task 4: Optional:

Edit subroutine output\_list. Use **field symbols or data references** instead of explicit work areas wherever you like.

1. Define suitably typed field symbols or data references.
2. Modify the LOOP statements.
3. Replace the respective structure with the field symbol or data reference.

## Solution 11: Field Symbols and Data References with Internal Tables

### Task 1:

Copy program BC402\_IDS\_ITAB\_TECH (or your own program, ZBC402\_##\_ITAB\_TECH) to the name **ZBC402\_##\_FSYM\_DREF**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program.
  - a) Carry out this step in the usual manner.

### Task 2:

Only edit subroutine get\_sums at first. Where it makes sense, use **data references** instead of explicit work areas.

1. Examine the source code of the subroutine. For which of the two internal tables can the work area be replaced completely by a data reference?

**Answer:** Only for internal table pt\_bookings. Due to the write access to ct\_sums (COLLECT statement), structure ls\_sums cannot be replaced by a data reference.

2. Define a suitably typed data reference.
  - a) See the source code excerpt from the model solution.
3. Modify the LOOP statement such that the data reference is filled during each loop pass, and not the work area.
  - a) See the source code excerpt from the model solution.
4. Replace structure ls\_bookings with the data reference and then deactivate (flag as comments) the definition of structure ls\_bookings.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

### Task 3:

Now edit subroutine get\_travelags. Where it makes sense, use **field symbols** instead of explicit work areas.

1. Examine the source code of the subroutine. For which of the two internal tables can the work area be replaced completely by a field symbol?  
**Answer:** Only for internal table pt\_bookings. Due to the write access to ct\_travelags (APPEND statement), structure ls\_travelags cannot be replaced by a field symbol.
2. Define a suitably typed field symbol.
  - a) See the source code excerpt from the model solution.
3. Modify the LOOP statement such that the current line is assigned to the field symbol during each loop pass, and not copied to the work area.
  - a) See the source code excerpt from the model solution.
4. Replace structure ls\_bookings with the field symbol and deactivate (flag as comments) the definition of structure ls\_bookings.
  - a) See the source code excerpt from the model solution.

### Task 4: Optional:

Edit subroutine output\_list. Use **field symbols or data references** instead of explicit work areas wherever you like.

1. Define suitably typed field symbols or data references.
  - a)
2. Modify the LOOP statements.
  - a) See the source code excerpt from the model solution.
3. Replace the respective structure with the field symbol or data reference.
  - a) See the source code excerpt from the model solution.

### Result

Source code excerpt from the model solution:

#### **Subroutine get\_sums**

```
* &-----*
*&      Form  GET_SUMS
*&-----*
```

*Continued on next page*

```

FORM get_sums USING      pt_bookings TYPE gty_t_bookings
              CHANGING ct_sums      TYPE gty_t_sums.

DATA:
*   ls_bookings LIKE LINE OF pt_bookings,
    lr_bookings TYPE REF TO  bc402_scus_book,
    ls_sums      LIKE LINE OF ct_sums.

LOOP AT pt_bookings REFERENCE INTO lr_bookings.

MOVE-CORRESPONDING lr_bookings->* TO ls_sums.
COLLECT ls_sums INTO ct_sums.

ENDLOOP.

ENDFORM.          " GET_SUMS

```

### Subroutine **get\_travelags**

```

*&-----*
*&      Form  GET_TRAVELAGS
*&-----*
FORM get_travelags USING      pt_bookings  TYPE gty_t_bookings
              CHANGING ct_travelags TYPE gty_t_travelags.

DATA:
*   ls_bookings  LIKE LINE OF pt_bookings,
    ls_travelags LIKE LINE OF ct_travelags.

FIELD-SYMBOLS:
  <fs_bookings>  LIKE LINE OF pt_bookings.

LOOP AT pt_bookings ASSIGNING <fs_bookings>.

MOVE-CORRESPONDING <fs_bookings> TO ls_travelags.
APPEND ls_travelags TO ct_travelags.

ENDLOOP.

SORT ct_travelags BY agencynum.

DELETE ADJACENT DUPLICATES FROM ct_travelags

```

*Continued on next page*

```

        COMPARING agencynum.

        SORT ct_travelags BY name.

ENDFORM.          " GET_TRAVELAGS

```

### Optional: Subroutine output\_list

```

*&-----*
*&      Form  OUTPUT_LIST
*&-----*

FORM output_list USING ps_customer  TYPE scustom
                  pt_bookings  TYPE gty_t_bookings
                  pt_sums       TYPE gty_t_sums
                  pt_travelags TYPE gty_t_travelags.

*  DATA:
*    ls_bookings  LIKE LINE OF pt_bookings,
*    ls_sums       LIKE LINE OF pt_sums,
*    ls_travelags LIKE LINE OF pt_travelags.

FIELD-SYMBOLS:
  <fs_bookings>  LIKE LINE OF pt_bookings,
  <fs_sums>       LIKE LINE OF pt_sums,
  <fs_travelags>  LIKE LINE OF pt_travelags.

DATA lv_text TYPE string.

CONCATENATE text-wcm
              ps_customer-form
              ps_customer-name
              INTO lv_text
              SEPARATED BY space.
CONDENSE lv_text.

WRITE: / lv_text.
SKIP.

WRITE / text-lob.
ULINE.
SKIP.

```

*Continued on next page*

```
LOOP AT pt_bookings ASSIGNING <fs_bookings>.  
  WRITE: /  
    <fs_bookings>-bookid,  
    <fs_bookings>-carrid RIGHT-JUSTIFIED,  
    <fs_bookings>-connid,  
    <fs_bookings>-fldate,  
    <fs_bookings>-cityfrom RIGHT-JUSTIFIED,  
    '->',  
    <fs_bookings>-cityto,  
    <fs_bookings>-forcuram CURRENCY -forcurkey,  
    <fs_bookings>-forcurkey.  
  ENDLOOP.  
  SKIP 2.  
  
  WRITE: / text-sum.  
  ULINE.  
  SKIP.  
  
  LOOP AT pt_sums ASSIGNING <fs_sums>.  
    WRITE: /  
      <fs_sums>-forcuram CURRENCY <fs_sums>-forcurkey,  
      <fs_sums>-forcurkey.  
    ENDLOOP.  
    SKIP 2.  
  
  WRITE / text-tag.  
  ULINE.  
  SKIP.  
  
  LOOP AT pt_travelags ASSIGNING <fs_travelags>.  
    WRITE: /  
      <fs_travelags>-name,  
      <fs_travelags>-city.  
    ENDLOOP.  
  
  ENDFORM.          " OUTPUT_LIST
```



## Lesson Summary

You should now be able to:

- Explain the syntax for working with data references and field symbols
- Use field symbols and data references to access the content of internal tables



## Unit Summary

You should now be able to:

- Explain the differences between integer, floating-point, and fixed-point arithmetic
- Explain how the runtime environment evaluates arithmetic expressions
- Understand Unicode and the stricter syntax requirements it imposes
- Use operators and functions to analyze byte strings and character strings
- Use string templates to define character strings
- Use regular expressions to define search patterns
- Describe the differences between standard, sorted, and hashed table types
- Tell the difference between key accesses and index accesses to internal tables
- Use the different table types correctly
- Use the BINARY SEARCH addition correctly
- Use the DELETE ADJACENT DUPLICATES and COLLECT statements correctly
- Describe the function of secondary keys for internal tables and apply them correctly.
- Explain the syntax for working with data references and field symbols
- Use field symbols and data references to access the content of internal tables



Internal Use SAP Partner Only

# *Unit 4*

## **Dynamic Programming**

### **Unit Overview**

ABAP features a number of options for dynamic program design. We discuss the most important of them in this unit. Aside from full generation of entire programs, this primarily involves dynamic components of statements, such as database accesses and calls. You learn how to use data references and field symbols to access data objects dynamically, as well as determine information about data types and data objects at runtime. Lastly, you see how to generate data objects and data types dynamically at runtime.



### **Unit Objectives**

After completing this unit, you will be able to:

- Outline the various techniques for dynamic programming available in ABAP
- Dynamically define ABAP syntax components (tokens)
- Define parts of SELECT statements at runtime
- Call function modules dynamically
- Call transactions and programs dynamically
- Generate a complete program dynamically
- Explain what generic types are and what they are used for
- Use generically typed parameters, field symbols, and data references
- Use field symbols to access data objects dynamically
- Use field symbols to access attributes and structure components dynamically
- Query the properties of data objects and data types at runtime
- Query the properties of classes and instances at runtime
- Generate objects (instances) at runtime
- Generate data objects at runtime
- Generate data types at runtime

### **Unit Contents**

Lesson: Overview.....	255
Lesson: Dynamic Calls and Program Generation .....	258

Exercise 12: Dynamic Method Calls .....	269
Lesson: Generic Data Types and Dynamic Access to Data Objects.....	275
Exercise 13: Dynamic SELECT Statements .....	287
Exercise 14: Generic Data Types and Dynamic Access to Structure Components.....	295
Lesson: Querying Type Attributes at Runtime .....	304
Exercise 15: Optional: Column Headers with Runtime Type Identification .....	319
Lesson: Generating Objects, Data Objects, and Data Types at Runtime	329
Exercise 16: Generating Data Objects at Runtime .....	339
Exercise 17: Optional: Generating Data Objects at Runtime with Runtime Type Creation (RTTC).....	347

# Lesson: Overview

## Lesson Overview

This lesson provides you with a brief overview of the different dynamic programming techniques supported in ABAP.



## Lesson Objectives

After completing this lesson, you will be able to:

- Outline the various techniques for dynamic programming available in ABAP

## Business Example

You have been asked to develop a flexible application. You first want to find out about the different dynamic techniques that are available in the ABAP programming language.

## Dynamic ABAP Programming



### Dynamic statements

```
gv_tabname = 'SFLIGHT'.
SELECT * FROM (gv_tabname).
```

### Generating programs from within programs



### Field symbols



### Data (type) generation at runtime

```
CREATE DATA gr_itab TYPE HANDLE go_tabletype.
```

Figure 134: Overview of Dynamic Programming Techniques in ABAP

**Dynamic programming** means specifying the type, length, offset, number of lines, or name of a program object at runtime. In the extreme case, the program object itself is generated at runtime.

ABAP supports various kinds of dynamic programming:

- Dynamic extension of internal tables and strings
- Dynamic offset and length specification (in MOVE, WRITE TO, formal parameters, and field symbols)
- Dynamic specification of program object names (the name of a table, modularization unit, sort criterion, control level criterion, and so on)
- Dynamic specification of type and data declarations

You can use the following techniques:

#### **Dynamic language elements**

Certain statements allow dynamic programming. For example, you can specify a function module name at runtime in the CALL FUNCTION statement.

#### **Generating programs from within programs**

You can store ABAP code in an internal table and generate it into a program at runtime.

#### **Field symbols**

(Dynamic ASSIGN, dynamic type specification, ASSIGN COMPONENT)

#### **Runtime type services**

(Dynamic type generation and data object generation at runtime)



## Lesson Summary

You should now be able to:

- Outline the various techniques for dynamic programming available in ABAP

# Lesson: Dynamic Calls and Program Generation

## Lesson Overview

In this lesson, you learn how to specify components of ABAP statements dynamically - that is, at runtime. You learn how to use this technique to design dynamic database accesses, calls of modularization units, and program calls. Lastly, you learn how to actually generate programs at runtime, and not just generate dynamic calls.



## Lesson Objectives

After completing this lesson, you will be able to:

- Dynamically define ABAP syntax components (tokens)
- Define parts of SELECT statements at runtime
- Call function modules dynamically
- Call transactions and programs dynamically
- Generate a complete program dynamically

## Business Example

You have been asked to develop a flexible application. To do so, you need to design dynamic calls of modularization units and database accesses. You want to learn about techniques using dynamic syntax components (tokens) and learn how to use this technique in calls and database accesses.

## Defining Parts of ABAP Statements at Runtime

In many ABAP statements, parts of the statement do not have to be defined until runtime. To enable this, the corresponding part of the statement is replaced with a data object. The contents of the data object are evaluated at runtime and the statement is executed accordingly. We differentiate between two cases with regard to syntax, depending on whether the data object is used in place of a literal or an identifier.



Replacing literals with variables:

```
* static:
```

```
DO 5 TIMES.  
...  
ENDDO.
```

```
* dynamic:
```

```
lv_int = 5.  
DO lv_int TIMES.  
...  
ENDDO.
```

```
* static:
```

```
WRITE lv_string+2(5).
```

```
* dynamic:
```

```
lv_len = 5.  
lv_off = 2.  
WRITE lv_string+lv_off(lv_len).
```

```
* static:
```

```
MESSAGE e038(BC402).  
MESSAGE ID 'BC402'  
TYPE 'E'  
NUMBER '038'.
```

```
* dynamic:
```

```
lv_msgid = 'BC402'.  
lv_msgty = 'E'.  
lv_msgno = '038'.  
MESSAGE ID lv_msgid  
TYPE lv_msgty  
NUMBER lv_msgno.
```

**Figure 135: Tokens: Replacing Literals**

When you replace literals, the data object can appear in place of the literal without any special additions. Logically, numeric literals should be replaced by integer data objects, while text literals are replaced by character-type objects with type c or string.



**Hint:** In some cases, text literals can also be replaced by internal tables with character-type line types.



**Caution:** Many of the errors that the syntax check can recognize and stop with static specifications result in runtime errors when dynamic specification is used. Therefore, the developer has to catch and handle these runtime errors for dynamic specification accordingly, to ensure no program terminations occur. Example:

If the dynamic length/offset access in the above example exceeds the length of the character string, the catchable runtime error DATA\_OFFSET\_LENGTH\_TOO\_LARGE occurs (exception class CX\_SY\_RANGE\_OUT\_OF\_BOUNDS).



Replacing identifiers with variables in parentheses (no spaces!):

\* static:

```
SORT lt_spfli BY cityfrom.
```

\* dynamic:

```
lv_comp = 'CITYFROM'.
SORT lt_spfli BY (lv_comp).
```

\* static:

```
READ TABLE lt_spfli
  INTO ls_spfli
  WITH KEY carrid = 'LH'
    connid = '0400'.
```

\* dynamic:

```
lv_nam1 = 'CARRID'.
lv_val1 = 'LH'.
lv_nam2 = 'CONNID'.
lv_val2 = '0400'.
READ TABLE lt_spfli
  INTO ls_spfli
  WITH KEY (lv_nam1) = lv_val1
    (lv_nam2) = lv_val2.
```

**Figure 136: Tokens: Replacing Identifiers**

When you replace identifiers, you have to flag the data objects specifically or risk ambiguity (is it calling the contents of the data object or the data object itself?).

Therefore, the data object whose content appears in place of an identifier must be specified in parentheses.



**Hint:** Note that no spaces are allowed between the parentheses in this case.

The ABAP syntax must support the dynamic specification of certain components of an ABAP statement. For details, see the documentation for the respective statement.

## Dynamic Open SQL

An important application for tokens is the flexible design of database accesses with Open SQL. This is especially relevant for read access with SELECT, but statements for write accesses also offer this option.

You can select nearly all components of a SELECT statement with data objects - either a character-type elementary data object or an internal table with a character line type. The diagram below shows several examples.



Dynamic FROM clause:

```
lv_tabname = 'SPFLI'.
SELECT COUNT(*) FROM (lv_tabname)
INTO lv_int.
```

Internal tables with  
character-type row  
type also possible

Dynamic field list:

```
lv_fieldlist = `CARRID CONNID CITYFROM CITYTO`.
SELECT (lv_fieldlist) FROM spfli
INTO CORRESPONDING FIELDS OF TABLE lt_spfli.
```

Dynamic WHERE condition (one line):

```
lv_cond = `CARRID = 'LH' AND CONNID = '0400'`.
SELECT * FROM spfli INTO TABLE it_spfli
WHERE (lv_cond).
```

Dynamic WHERE condition (multiline):

```
APPEND `CARRID = 'LH' AND ` TO lt_cond.
APPEND `CONNID > '0400'` TO lt_cond.

SELECT * FROM spfli INTO TABLE it_spfli
WHERE (lt_cond).
```

**Figure 137: Dynamic Components of the SELECT Statement**

Aside from the FROM addition, the field list, and the WHERE clause, you can also use this option for the sort sequence (ORDER BY addition), and the GROUPING and HAVING additions. The INTO clause is the only one that does not support this dynamic technique.



**Hint:** Field symbols and data references are used to dynamically determine the data object that is filled by the SELECT statement. We learn about this technique in a later lesson.



**Caution:** If the dynamic values do not result in an SQL statement with correct semantics and syntax, runtime errors occur (exception classes CX\_SY\_DYNAMIC\_OSQL\_SEMANTICS and CX\_SY\_DYNAMIC\_OSQL\_SYNTAX). If you use dynamic SQL statements, be sure to always catch and handle these errors.

## Dynamic Calls

ABAP enables you to call modularization units and programs dynamically. In this approach, the name of the modularization unit, the transaction, or the program is replaced by a character-type data object in the call syntax.



Subroutine:

```
PERFORM (lv_form_name) IN PROGRAM (lv_prog_name) ... .
```

Function module:

```
CALL FUNCTION lv_func_name ...
```

Method:

```
CALL METHOD lo_instance->(lv_meth_name) ...  
CALL METHOD (lv_class_name)=>(lv_static_meth_name) ...
```

Executable program:

```
SUBMIT (lv_report_name) ....
```

Transaction:

```
CALL TRANSACTION lv_transaction_code ...  
LEAVE TO TRANSACTION lv_transaction_code ...
```

**Figure 138: Dynamic Calls - Overview**

The following simple rule applies:

**No parentheses** for **function modules** or **transaction codes**, because variables are replacing text literals here.

**Parentheses** for **subroutines**, **methods**, and **executable programs**. This involves the replacement of identifiers.

You can **pass parameters** in dynamic calls statically or dynamically. Of course, specifying static parameters only makes sense if you only want to call modularization units or programs with the same interface. The sections below discuss the syntax for specifying dynamic parameters for function modules, methods, and reports.

→ **Note:** Since parameters are always passed dynamically in transaction calls (specifying a table with the USING addition), no special handling is needed here.

## Dynamic Call of Function Modules

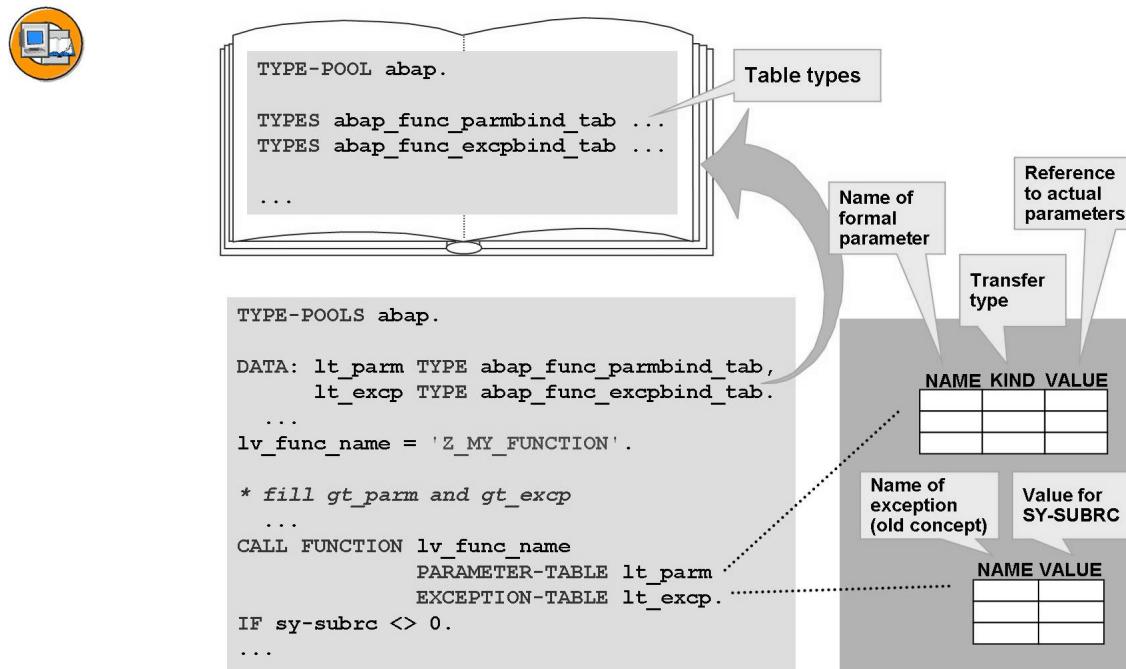


Figure 139: Dynamic Call of Function Modules

You can assign the actual parameters to the formal parameters dynamically in function module calls, by specifying an appropriately typed internal table after the PARAMETER-TABLE statement. You cannot combine the PARAMETER-TABLE addition with IMPORTING, EXPORTING, CHANGING, or TABLES.

The internal table must have table type ABAP\_FUNC\_PARMBIND\_TAB from type group ABAP. The line type of this table type has the following components:

### NAME

Name of formal parameter.

Note: Must be specified in all caps.

### KIND

Type of parameter transfer (exporting, importing, changing, or tables).

This is an integer type field. The ABAP type group provides the constants `abap_func_exporting`, `abap_func_importing`, and so on, to fill it. Note that a parameter that is defined as an export parameter in the module must have a set constant `abap_func_importing` (and vice versa).

## VALUE

Data reference to actual parameter.

This field has the generic type TYPE REF TO DATA, which means it can point to any type of data object. The easiest thing is to fill the field with a GET REFERENCE OF `act_par` INTO `ls_partab-value` statement. `act_par` stands for the actual parameter and `ls_partab` stands for the work area of the parameter table.



**Caution:** Table type ABAP\_FUNC\_PARMBIND\_TAB defines a **sorted** table - so avoid using the APPEND statement to fill it. Instead, use key access with INSERT ... INTO TABLE.

If the function module raises **conventional exceptions**, they can also be caught dynamically. To do so, you specify an internal table with type ABAP\_FUNC\_EXCPBIND\_TAB after the EXCEPTION-TABLE edition, in which the NAME and VALUE columns define the assignment between the conventional exception and a value for system field sy-subrc. In this case, the evaluation of sy-subrc has to be implemented (statically) after the dynamic call of the function module, as usual.



**Caution:** If errors occur in the tables or in the dynamic call of the function module itself, catchable runtime errors occur (exception classes CX\_SY\_DYN\_CALL\_...). Be sure to catch and handle these errors.



**Hint:** Instead of formulating a separate CATCH block for each exception class, you can catch and handle the shared superclass, CX\_SY\_DYN\_CALL\_ERROR, instead.

## Dynamic Method Call

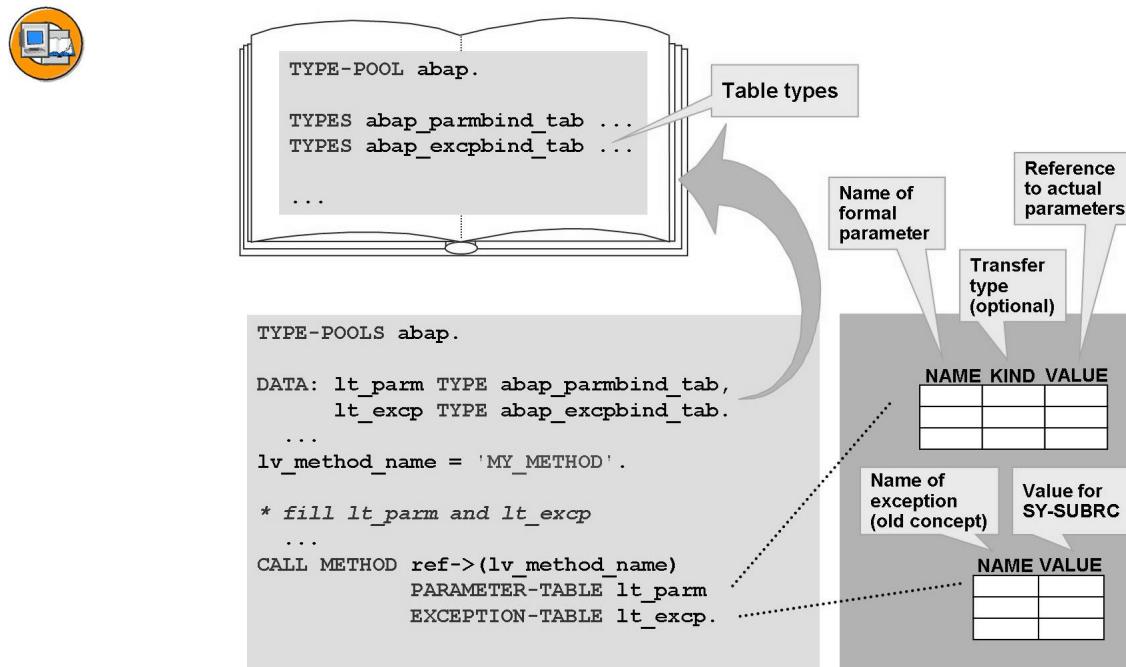


Figure 140: Dynamic Method Call

In method calls, the dynamic passing of parameters and handling of conventional exceptions is similar to the procedure for function modules. Still, there are a number of minor differences:

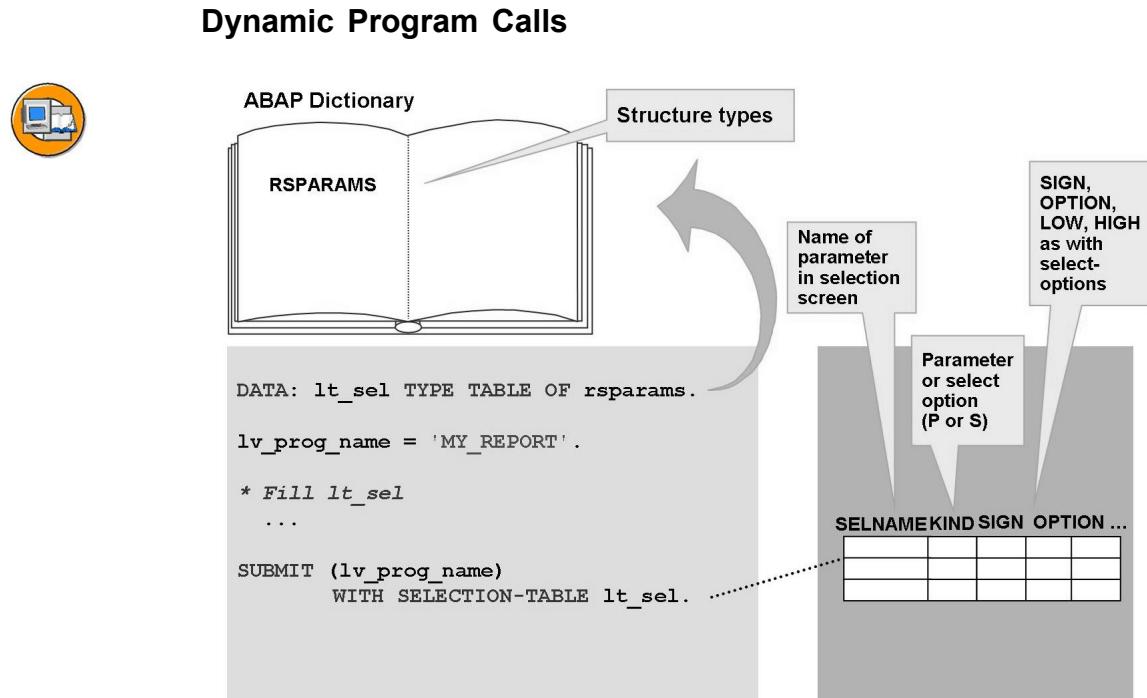
The internal tables must have table type ABAP\_PARMBIND\_TAB and ABAP\_EXCP\_TAB from type group ABAP.

Table type ABAP\_PARMBIND\_TAB describes a hashed table, not a sorted table. Therefore, the syntax check prevents index accesses.

The KIND field can remain initial in method calls. If you want to fill the field anyway, the corresponding constants are contained in global class CL\_ABAP\_OBJECTDESCR, not type group ABAP.



**Caution:** Dynamic method calls can also result in easily catchable runtime errors if the dynamic values are incorrect. The exception classes for these runtime errors are also subclasses of class CX\_SY\_DYN\_CALL\_ERROR.



**Figure 141: Dynamic Call of an Executable Program (Report)**

When you use dynamic calls of executable programs, you can supply the selection screen with values dynamically by passing an internal table with the WITH SELECTION-TABLE addition. This internal table must be defined with Dictionary type RSPARAMS as its line type. The RSPARAMS structure type has the following components:

#### SEL\_NAME

Specify the name of the parameter or selection criterion (SELECT-OPTIONS) on the selection screen. The name must be in all caps.

#### KIND

The values “P” and “S” determine whether it is a parameter or a selection criterion.

#### SIGN, OPTION, LOW, HIGH

For selection criteria (KIND = 'S'), these values have the same meanings as the corresponding columns of a selection table. For parameters (KIND = 'P'), only the LOW field has to be filled.

## Generating Dynamic Programs

In some cases, it can be necessary to generate entire programs permanently or transiently, as this is the only way to achieve the required dynamism. In this section, we briefly discuss the two options for dynamic program generation.

## Generating Persistent Programs



```
INSERT REPORT <program> FROM <itab>.  
SUBMIT <program> [AND RETURN].
```

```
DATA gt_tab TYPE TABLE OF string.  
DATA gv_prg TYPE program.  
  
*Fill internal Table  
APPEND 'REPORT ztest.' TO gt_tab.  
APPEND 'PARAMETERS pa type s_carr_id.' TO gt_tab.  
APPEND 'START-OF-SELECTION.' TO gt_tab.  
APPEND 'WRITE ''Hello''. ' TO gt_tab.
```

```
gv_prg = 'ZTEST'.
```

```
INSERT REPORT gv_prg FROM gt_tab.
```

```
SUBMIT (gv_prg) VIA SELECTION-SCREEN  
AND RETURN.
```



**Figure 142: Generating Persistent Executable Programs**

The `INSERT REPORT` statement creates a program that is stored permanently in the repository. To do so, you fill in an internal table with the source code, the first line of which is the `REPORT` statement. Do not forget the period at the end of each ABAP statement.

The contents of the internal table are saved as an executable program (program type *I*). Once the program has been saved, you can start it with the `SUBMIT` statement. The `VIA SELECTION-SCREEN` addition causes the selection screen (if any) to be displayed. The `AND RETURN` addition means control passes back to the calling program after the program ends.

For more information about generating programs, refer to the ABAP help for keyword `INSERT REPORT`.



## Generating Transient Subroutine Pools

```
GENERATE SUBROUTINE POOL <itab> NAME <program>.  
PERFORM <form_name> in program <program>.
```

```
DATA gt_tab TYPE TABLE OF string.  
DATA gv_prg TYPE program.  
  
*Fill internal Table  
APPEND 'REPORT ztest.'      TO gt_tab.  
APPEND 'FORM show.'         TO gt_tab.  
APPEND 'WRITE ''Hello'''.' TO gt_tab.  
APPEND 'ENDFORM.'           TO gt_tab.
```

```
GENERATE SUBROUTINE POOL gt_tab  
      NAME gv_prg.
```

```
PERFORM show  
      IN PROGRAM (prog) IF FOUND.
```

Transient  
program

Figure 143: Generating Transient Subroutine Pools

The GENERATE SUBROUTINE POOL statement generates a program with type *S*, a **subroutine pool**. This program exists only at runtime of the generating program and can only contain subroutines. You call these subroutines with an external subroutine call (“external PERFORM”).

For more information, see the ABAP help for keyword *GENERATE*.

## Exercise 12: Dynamic Method Calls

### Exercise Objectives

After completing this exercise, you will be able to:

- Call methods dynamically
- Pass parameters on dynamically

### Business Example

You have been asked to develop a simple program that can display the contents of any database table. To do so, you want to find out about the options available in dynamic programming. You start with a (static) program that can display the contents of database table SPFLI or SCUSTOM. You then generalize the coding, step by step, until it is no longer specific to these two database tables, but instead can be used for any database table.

In the first step, replace two static method calls with a dynamic call.

#### Template:

BC402\_DYT\_DYN\_CALL

#### Solution:

BC402\_DYS\_DYN\_CALL

### Task 1:

Copy program BC402\_DYT\_DYN\_CALL to the name  
**ZBC402\_##\_DYN\_CALL**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Activate and test the program. Analyze the source code. How is the data read from the database? Into which data objects? How is the data displayed?

---

---

---

### Task 2:

Replace the two calls of the write\_connections and write\_customers methods with a single call **after** the ENDCASE statement. Use the option of specifying the method name with a character-type data object. Use the PARAMETER-TABLE addition to implement the passing of parameters.

1. Define a character-type data object for the method name (suggested name: **gv\_methname**). Fill the data object with the respective method names in the WHEN blocks.
2. Define an internal table whose type allows it to be defined after the PARAMETER-TABLE addition in the dynamic method call (suggested name: **gt\_parmbind**). Also define a work area for it (suggested name: **gs\_parmbind**).
3. Fill **gv\_methname** with the respective method names in the WHEN blocks.

Fill **gt\_parmbind** with one line each for passing the parameters in the WHEN blocks. Fill the *name* column with the name of the formal parameter for the respective method and point the reference in the *value* column to the internal table you want to pass on as actual parameters.



**Hint:** You can leave the *kind* column initial in the method call.

4. Move one of the two method calls from the WHEN block to after the ENDCASE statement and delete the other one.

Make the remaining method call dynamic: Replace the static method name with data object **gv\_methname** and replace the static parameter passing with the PARAMETER-TABLE addition.

### Task 3: Optional:

Catch the specific runtime errors that can occur during dynamic method calls.

1. Read the keyword documentation to find out about the exceptions that can occur during dynamic method calls. Implement exception handling with TRY... CATCH ... ENDTRY. Respond with error message 060 from message class BC402, for example

## Solution 12: Dynamic Method Calls

### Task 1:

Copy program BC402\_DYT\_DYN\_CALL to the name **zBC402##\_DYN\_CALL**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program. Analyze the source code. How is the data read from the database? Into which data objects? How is the data displayed?

**Answer:** The data is read from the database in static SELECT statements. In the process, one of two statically defined internal tables is filled. The output is implemented by calling the corresponding methods.

### Task 2:

Replace the two calls of the write\_connections and write\_customers methods with a single call **after** the ENDCASE statement. Use the option of specifying the method name with a character-type data object. Use the PARAMETER-TABLE addition to implement the passing of parameters.

1. Define a character-type data object for the method name (suggested name: **gv\_methname**). Fill the data object with the respective method names in the WHEN blocks.
  - a) See the source code excerpt from the model solution.
2. Define an internal table whose type allows it to be defined after the PARAMETER-TABLE addition in the dynamic method call (suggested name: **gt\_parmbind**). Also define a work area for it (suggested name: **gs\_parmbind**).
  - a) See the source code excerpt from the model solution.
3. Fill gv\_methname with the respective method names in the WHEN blocks.

Fill gt\_parmbind with one line each for passing the parameters in the WHEN blocks. Fill the *name* column with the name of the formal parameter for the respective method and point the reference in the *value* column to the internal table you want to pass on as actual parameters.



**Hint:** You can leave the *kind* column initial in the method call.

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

*Continued on next page*

4. Move one of the two method calls from the WHEN block to after the ENDCASE statement and delete the other one.

Make the remaining method call dynamic: Replace the static method name with data object gv\_methname and replace the static parameter passing with the PARAMETER-TABLE addition.

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

### Task 3: Optional:

Catch the specific runtime errors that can occur during dynamic method calls.

1. Read the keyword documentation to find out about the exceptions that can occur during dynamic method calls. Implement exception handling with TRY... CATCH ... ENDTRY. Respond with error message 060 from message class BC402, for example
  - a) See the source code excerpt from the model solution.

### Result

Source code excerpt from the model solution:

```
REPORT  bc402_dys_dyn_call MESSAGE-ID bc402.

TYPE-POOLS: abap.

DATA: gt_cust      TYPE ty_customers,
      gt_conn     TYPE ty_connections.

DATA: gv_methname TYPE string,
      gt_parmbind TYPE abap_parmbind_tab,
      gs_parmbind TYPE abap_parmbind.

SELECTION-SCREEN COMMENT 1(80) text-sel.
PARAMETERS:
  pa_xconn  TYPE xfeld RADIOBUTTON GROUP tab DEFAULT 'X',
  pa_xcust  TYPE xfeld RADIOBUTTON GROUP tab .
PARAMETERS:
  pa_nol    TYPE i DEFAULT '100'.

START-OF-SELECTION.

* specific part
*-----*
CASE 'X'.
```

*Continued on next page*

```

WHEN pa_xconn.

SELECT * FROM spfli INTO TABLE gt_conn
    UP TO pa_nol ROWS.

gv_methname = 'WRITE_CONNECTIONS'.

gs_parmbind-name = 'IT_CONN'.
GET REFERENCE OF gt_conn INTO gs_parmbind-value.
INSERT gs_parmbind INTO TABLE gt_parmbind.

WHEN pa_xcust.

SELECT * FROM scustom INTO TABLE gt_cust
    UP TO pa_nol ROWS.

gv_methname = 'WRITE_CUSTOMERS'.

gs_parmbind-name = 'IT_CUST'.
GET REFERENCE OF gt_cust INTO gs_parmbind-value.
INSERT gs_parmbind INTO TABLE gt_parmbind.

ENDCASE.

* dynamic part
*-----*

TRY.
CALL METHOD cl_bc402_utilities=>(gv_methname)
    PARAMETER-TABLE
        gt_parmbind.

CATCH cx_sy_dyn_call_error.
    MESSAGE e060.
ENDTRY.

```



## Lesson Summary

You should now be able to:

- Dynamically define ABAP syntax components (tokens)
- Define parts of SELECT statements at runtime
- Call function modules dynamically
- Call transactions and programs dynamically
- Generate a complete program dynamically

# Lesson: Generic Data Types and Dynamic Access to Data Objects

## Lesson Overview

In this lesson, you learn how you can use generic data types in ABAP for dynamic programming. You learn how you can use generically typed interface parameters and field symbols, for example, to define the target range in the INTO clause dynamically in dynamic SELECT statements. You learn about the constraints posed on generic data references by the ABAP syntax, as well as how they affect their use. You also learn how to use generic field symbols to access data objects dynamically.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain what generic types are and what they are used for
- Use generically typed parameters, field symbols, and data references
- Use field symbols to access data objects dynamically
- Use field symbols to access attributes and structure components dynamically

## Business Example

You have been asked to develop a flexible application. To do so, you want to find out about the generic data types in ABAP and learn how to type parameters, field symbols, and data references generically.

## Generic Data Types

In addition to the complete data types, ABAP also features a number of generic data types. In contrast to the complete types, generic types cannot be used to define the properties of data objects. Their use is limited to typing formal parameters and field symbols.

 **Note:** Currently, only the two built-in types ... TYPE REF TO OBJECT and ... TYPE REF TO DATA are available for typing reference variables generically. Therefore, we examine generically typed reference variables a bit later.

The ABAP syntax supports the following predefined generic data types:



Generic data type	Meaning	Compatible ABAP types
Fully generic any data	Any type Any data type	Any Any
Elementary, flat simple	Elementary data type or flat, character-type structure	Any
Numeric numeric decfloat	Numeric data type Numeric data type	i, f, p, decfloat16, decfloat32 decfloat16, decfloat32
Character-type clike csequence	Character data type Text data type	c, n, d, t, string c, string
Hexadecimal xsequence	Byte data type	x, xstring

Figure 144: Generic ABAP Data Types



Generic data type	Meaning	Compatible ABAP types
Table-type any table	Any internal table	Any table type
index table	Any index table	Standard or sorted
standard table	Any standard table	Only standard
sorted table	Any sorted table	Only sorted
hashed table	Any hashed table	Only hashed

Figure 145: Generic ABAP Data Types - Internal Tables



**Hint:** The generic types clike, csequence, numeric, simple, and xsequence are available as of *SAP Web AS 6.10*.



**Hint:** The generic type decfloat is available as of *SAP NetWeaver 7.0 EhP 2*.

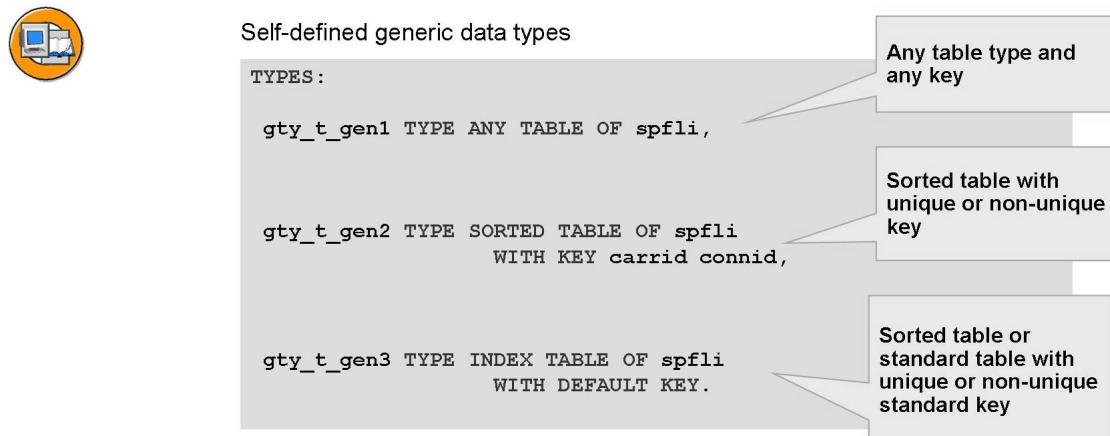
Each of the generic types is compatible with the set of ABAP types shown in the diagram above. When a procedure is called, the system checks whether the type of the actual parameter is compatible with the type of the formal parameter. When a generically typed field symbol is assigned, the syntax check tests whether the type of the data object is compatible with the generic type of the field symbol.

 **Note:** The DATA type behaves exactly the same as ANY. The differentiation is purely theoretical at this point. In future, however, reference variables with type TYPE REF TO ANY might be possible, to point to data objects or instances of classes.

The ANY TABLE, INDEX TABLE, SORTED TABLE, and other types are special, generic data types that are only compatible with internal tables. You can also restrict them to internal tables with a specific table type. The line type and key definition remain open, however.

In addition to these built-in generic table types, ABAP also has generic table types that developers can create as generic types - either in the *Data Dictionary* or in the program source code, with the TYPES statement.

The following diagram shows an example of locally defined generic table types:



**Figure 146: Generic Table Types - User-Defined**

In these generic table types, the line type is defined, but the table type and/or key definition remain open.

 **Note:** Appropriate generic types are created in the *Data Dictionary* by checking the corresponding options under *Access* and *Key*.

Aside from the generic table types, there are no user-defined generic table types in ABAP.

## Generically Typed Parameters

When formal parameters of a procedure (subroutine, function module, or method) are typed generically, the procedure becomes more flexible. Of course, you have to make sure that the procedure can actually handle the different types.

As mentioned above, the syntax check makes sure that only actual parameters with compatible types are passed on to generically typed formal parameters. When generically typed parameters are addressed in ABAP statements, however, you often cannot determine whether the type of the underlying data object or its current content is logical in the respective operand position until runtime. If conflicts occur, the system raises runtime errors that **cannot be caught** in most cases. The following diagram shows an example.



Method definition with fully generic parameters:

```
CLASS-METHODS write_any_table
  IMPORTING
    ig_info TYPE ANY
    it_data TYPE ANY.
```

Method implementation (with errors):

```
METHOD write_any_table.
  FIELD-SYMBOLS <ls_line> TYPE ANY.
  LOOP AT it_data ASSIGNING <ls_line>.
  ...
  ENDLOOP.
  WRITE / ig_info.
ENDMETHOD.
```

Syntax error!  
Only generic table types  
are allowed in this  
operand position!

Possible runtime error!  
Only elementary data objects or flat  
character-type structures are allowed  
in these operand positions!

**Figure 147: Problems When Parameters Are Too General**

The WRITE statement throws an uncatchable exception, OBJECTS\_NOT\_CHAR-CONV, if the parameter is given an internal table or a structure that is not flat and character-type.

The syntax check helps avoid such runtime errors particularly in operand positions that only allow internal tables (such as SELECT ... INTO TABLE ..., LOOP AT TABLE ..., and so on): It only accepts generic table types in these operand positions. Therefore, you have to make sure an internal table comes after the formal parameter.

To avoid syntax errors and runtime errors, select generic types that are specific enough to be used with all compatible types in the desired operand position.



Method definition with matching generic parameters:

```
CLASS-METHODS write_any_table
  IMPORTING
    ig_info TYPE SIMPLE
    it_data TYPE ANY TABLE.
```

Implementing methods:

```
METHOD write_any_table.
  FIELD-SYMBOLS <ls_line> TYPE ANY.

  LOOP AT it_data ASSIGNING <ls_line>.
    ...
  ENDLOOP.

  WRITE / ig_info.

ENDMETHOD.
```

No syntax error  
The generic ANY TABLE type ensures that the data object is an internal table

No danger of runtime errors  
The generic SIMPLE type only allows data objects that are permitted at this point

**Figure 148: Avoiding Errors by Typing as Precisely as Possible**

If you use the generic type SIMPLE, the WRITE statement cannot cause runtime errors. All data types that are compatible with SIMPLE can be converted to a character string.

The syntax errors are corrected because parameter it\_data now only accepts internal tables, thanks to its type ANY TABLE.



**Hint:** A runtime error could still occur if the method contains an index access to it\_data, however - specifically, if the actual parameter is a hashed table. In this case, you should use the generic type INDEX TABLE to restrict the formal parameter even further.

## Generically Typed Field Symbols

Field symbols are pointers that can be assigned to data objects dynamically. When the field symbol is used in an ABAP statement, you can dynamically define the specific data object that the field symbol refers to.

If the field symbol is fully typed, all potential data objects must have the same type, which restricts the possible uses of field symbols significantly.

### Using Generically Typed Field Symbols

Generic typing enables the field symbol to refer to data objects with different types. As a result, you can address different data objects in the same operand position of an ABAP statement. The following diagram shows an example:



```

DATA: lt_scarr TYPE TABLE OF scarr,
      lt_sbook TYPE TABLE OF sbook.

FIELD-SYMBOLS:
  <fs_tab> TYPE ANY TABLE.

CASE lv_table_name.
  WHEN 'SCARR'.
    ASSIGN lt_scarr TO <fs_tab>.

  WHEN 'SBOOK'.
    ASSIGN lt_sbook TO <fs_tab>.

ENDCASE.

IF <fs_tab> IS ASSIGNED.
  SELECT * FROM (lv_table_name)
    UP TO 100 ROWS
    INTO TABLE <fs_tab>.

ENDIF.

```

**Field symbols with ANY TABLE type can point to any internal tables**

**An internal table that fits the name of the database table is selected**

**The field symbol is allowed at this point because it always points to an internal table**

**Figure 149: Example: Generic Field Symbol in Dynamic SELECT**

You want the program to access one of two database tables, SCARR or SBOOK, depending on the content of data object lv\_table\_name. To do so, you program a dynamic SELECT statement. To ensure the line type of the internal table matches the field list of the database table, a generically typed field symbol is used as a target that points to internal table lt\_scarr or lt\_sbook as needed.



**Hint:** The field symbol cannot have type ANY. While this type could be assigned to the two internal tables without any problems, it would cause a syntax error when used in the SELECT.



**Note:** Of course, this example is only somewhat dynamic, since the data objects used already have to be defined statically. You learn how to generate data objects at runtime in a later lesson.

## Type Casting for Field Symbols



```
ASSIGN data_object TO <fs> CASTING [ TYPE type_name | ... ] .
```

```
TYPES: BEGIN OF gty_s_date,
         year  TYPE n LENGTH 4,
         month TYPE n LENGTH 2,
         day   TYPE n LENGTH 2,
       END OF gty_s_date.

* option 1: implicit
FIELD-SYMBOLS <fs> TYPE gty_s_date.

ASSIGN sy-datum TO <fs> CASTING.
WRITE: / <fs>-year, <fs>-month, <fs>-day.

* option 2: explicit
FIELD-SYMBOLS: <fs> TYPE ANY.
...
ASSIGN sy-datum TO <fs> CASTING TYPE gty_s_date.
...
```

Content of assigned data object is interpreted as if it had the implicitly or explicitly specified type

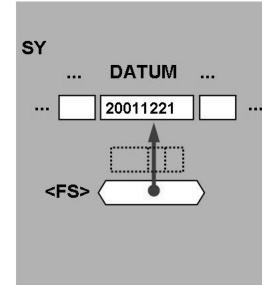


Figure 150: Type Casting for Field Symbols

If you use the CASTING addition when you assign a data object to a field symbol that has a different type, you can remove the restrictions of having to use the data object's original type. The data object is then interpreted **as though** it had the data type of the field symbol.

If you use the CASTING TYPE addition when you assign a data object to a field symbol that has a different type, you can access the data object using the field symbol as if the object had that **explicitly specified** type.

In the above example, note that the system field sy-datum is an elementary character-type component of length 8.

## Dynamic Access to Data Objects using Field Symbols

In most ABAP statements, data objects in operand positions cannot be replaced directly by character-type data objects in parentheses. This option is available in the ASSIGN statement for the data object to which the field symbol is to be assigned, however:



Any data object:

```
lv_name = 'LV_CARRID'.
ASSIGN (lv_name) TO <fs>.
```

Structure component:

```
lv_name = 'LS_SPFLI-CARRID'.
ASSIGN (lv_name) TO <fs>.
```

Static attribute:

```
lv_name = 'LCL_VEHICLE=>N_O_AIRPLANES'.
ASSIGN (lv_name) TO <fs>.
```

6.10 and higher

Instance attribute:

```
lv_name = 'LO_VEHICLE->N_O_AIRPLANES'.
ASSIGN (lv_name) TO <fs>.
```

6.10 and higher

**Figure 151: Dynamic Access to Data Objects**

This approach makes it indirectly possible to address the data object dynamically by its name. You merely have to assign it a generically typed field symbol, using the syntax above, and then use that field symbol in the ABAP statement.

Easier, less error-prone alternatives are available specifically to access attributes and structure components. They are described below.

## Dynamic Access to Object Attributes and Class Attributes



Static attribute (using the full name):

```
lv_name = 'LCL_VEHICLE=>N_O_AIRPLANES'.
ASSIGN (lv_name) TO <fs>.
```

Instance attribute:

```
lv_attribut_name = 'MAKE'.
ASSIGN lo_vehicle->(lv_attribut_name) TO <fs>.
```

Attribute name in uppercase or lowercase.

Static attribute:

```
lv_attribut_name = 'N_O_AIRPLANES'.
lv_class_name    = 'LCL_VEHICLE'.
ASSIGN (lv_class_name)=>(lv_attribut_name) TO <fs>.
```

Attribute name and class name in uppercase or lowercase.

**Figure 152: Dynamic Access to Object Attributes and Class Attributes**

In addition to access using the full identifier for dynamic access to attributes of classes an instance, another variant allows you to specify the attribute name solely with a character-type data object. The object reference and component selector are specified statically. The class name for static attributes can also be specified dynamically, if necessary. This syntax is similar to that for dynamic method calls.



**Hint:** Since the static part of the statement is supported by the syntax check, these variants are less error-prone and are therefore preferred.

## Dynamic Access to Structure Components



Structure component (using the full name):

```
lv_name = 'LS_SPFLI-CARRID'.  
ASSIGN (lv_name) TO <fs>.
```

Structure component (using its component name):

```
lv_comp_name = 'CARRID'.  
ASSIGN COMPONENT lv_comp_name  
      OF STRUCTURE ls_spfli      TO <fs>.
```

Structure component (using its position in the structure):

```
lv_comp_number = 2.  
ASSIGN COMPONENT lv_comp_number  
      OF STRUCTURE ls_spfli      TO <fs>.
```

**Figure 153: Dynamic Access to Structure Components**

A special variant of the ASSIGN statement is available for dynamic access to structure components. The structure is specified statically (or through a separate field symbol) in the ASSIGN COMPONENT ... OF STRUCTURE ... statement. The components can be addressed by name or by their position (number) within the structure.



**Hint:** Since the static part of the statement is supported by the syntax check, these variants are less error-prone and are therefore preferred.

Since the components of a structure are addressed by their positions in sequence, you can process any structure completely, component by component. The next figure shows an example:



```

CLASS-METHODS:
    write_any_struct
        IMPORTING is_struct TYPE any.

METHOD write_any_struct.

FIELD-SYMBOLS: <ls_comp> TYPE simple.

DO.

ASSIGN COMPONENT sy-index OF STRUCTURE is_struct
    TO <ls_comp>.

IF sy-subrc <> 0.
    EXIT.
ELSE.
    WRITE <ls_comp>.
ENDIF.

ENDDO.

ENDMETHOD.

```

**Any structure  
(with elementary components)**

**sy-subrc <> 0:  
No components with  
this number  
→ End of loop**

Figure 154: Full Processing of Any Non-Nested, Flat Structure



**Caution:** In the above example, an uncatchable ASSIGN\_TYPE\_CONFLICT runtime error occurs if the transferred structure contains a component that is incompatible with the generic SIMPLE type of the field symbols. (Likewise, a runtime error occurs during the WRITE statement if the field symbol is typed with ANY.)



**Note:** In the next lesson, you learn how you can determine the component type before the ASSIGN statement, to avoid runtime errors.

## Generically Typed Data References

Reference variables can also be typed generically. In contrast to field symbols, however, only the generic types DATA (for data references) and OBJECT (for object references) are available for reference variables.

The use of generically typed reference variables is highly restricted in ABAP. With the exception of the ASSIGN statement, generically typed cannot be dereferenced directly in operand positions. The statement WRITE ref->\* has incorrect syntax, for example, if the data reference has type TYPE REF TO DATA.

Due to these restrictions, generic data references are only used where generic field symbols do not work:

## Using Generic Data References

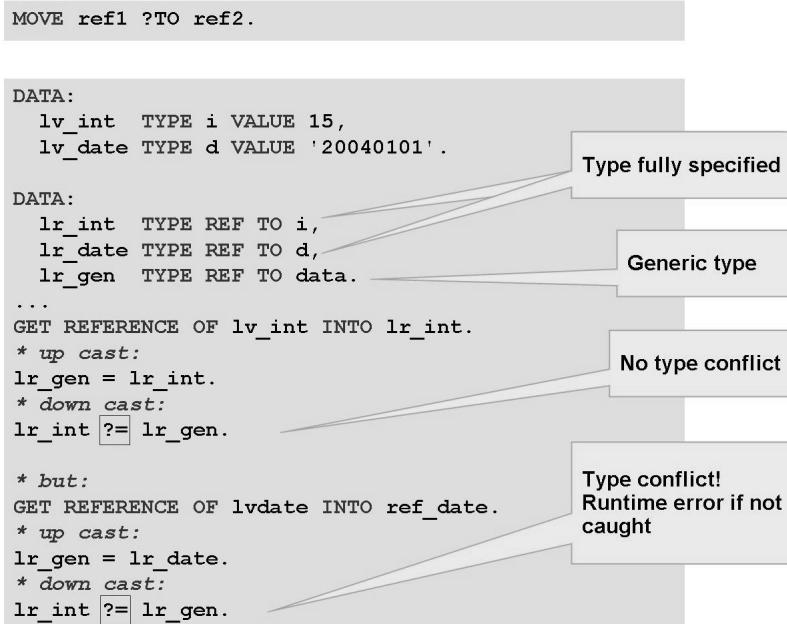


- In internal tables (see parameter table for dynamic procedure call)
- During the dynamic generation of data objects using the CREATE DATA statement (see next lesson).

There are two options for addressing the contents of the referenced data object for references with type TYPE REF TO DATA. We discuss both of them below.

### Cast Assignment for Data References

One option for addressing the content of the referenced data objects uses a fully typed data reference. The contents of the generic data reference are copied to the fully typed data reference. The complete data reference can then be dereferenced in any operand item. For this technique to work, however, the second reference variable must have the same exact type as the referenced data object.



**Figure 155: Cast Assignment for Data References**

When you assign values between two reference variables with different types, this is called a **cast assignment**. It is similar to cast assignment between object references, which you should be familiar with from object-oriented programming.

As with object references, an up cast is when the target variable has a more general definition and a down cast is when the target variable has a more specific type than the source variable. Accordingly, assignment from a generic data reference to a full reference is a down cast.

While up casts to type TYPE REF TO DATA always work, the down cast must have the ?= cast operator to have proper syntax. Compatibility is checked at runtime in this case.



**Caution:** If the type of the new reference variable does not match the exact type of the referenced data object, a runtime error occurs. You should catch it (exception class CX\_SY\_MOVE\_CAST\_ERROR).

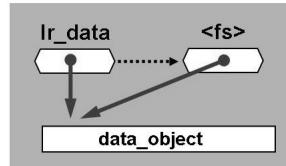
## Dereferencing Generically Typed Data References

The only ABAP statement in which a generic reference variable can be dereferenced is the ASSIGN statement.



Generically-typed data reference:

```
DATA lr_data TYPE REF TO DATA.  
...     " Fill the generic data reference  
ASSIGN lr_data->* TO <fs>.
```



- Generically typed data references can only be dereferenced in the ASSIGN statement in ABAP.
- After the ASSIGN statement, the (generic) field symbol points to the same data object as the data reference.
- The field symbol can be used to address the content of the referenced data object.

**Figure 156: Dereferencing Generically Typed Data References**

The ASSIGN statement assigns a (generic) field symbol to the data object to which the generic reference variable points. The contents of the referenced object can then be addressed and processed further using this field symbol.

## Exercise 13: Dynamic SELECT Statements

### Exercise Objectives

After completing this exercise, you will be able to:

- Define the database table in a SELECT statement dynamically
- Use a generically typed field symbol to specify the target range of a SELECT statement dynamically
- 

### Business Example

You have been asked to develop a simple program that can display the contents of any database table. To do so, you want to find out about the options available in dynamic programming.

In the next step, replace two static SELECT statements with a dynamic statement.

#### Template:

BC402\_DYS\_DYN\_CALL

#### Solution:

BC402\_DYS\_DYN\_SQL

#### Task 1:

Copy program BC402\_DYS\_DYN\_CALL (or your own program, ZBC402\_##\_DYN\_CALL) to the name **ZBC402\_##\_DYN\_SQL**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Activate and test the program. Analyze the source code. How is the data read from the database? Into which data objects?

---

---

---

### Task 2:

Replace the two SELECT statements with a single SELECT statement **after** the ENDCASE statement. Use the option of specifying the table name as a character-type data object. In addition, use a generically typed field symbol as the target range (INTO clause), which you dynamically assign a suitable data object.

1. Define a character-type data object for the table name (suggested name: **gv\_tabname**). Fill the data object with the respective table names in the WHEN blocks.
2. Create a field symbol (suggested name: **<fs\_table>**) whose type lets it be assigned either data object gt\_conn or gt\_cust.
3. Why is type ANY not suitable?

---

---

---

4. Assign the field symbol the respective internal tables in the WHEN blocks - the ones you want to fill in the generic SELECT statement.
5. Move one of the two SELECT statements from the WHEN block to after the ENDCASE statement and delete the other one.

Make the remaining database access dynamic: Replace the static table name with data object gv\_tabname and replace the static values of the target range with the value in the field symbol.

### Task 3: Optional:

Catch the specific runtime errors that can occur when using dynamic SQL statements.

1. Read the keyword documentation to find out about the exceptions that can occur when using dynamic SQL statements. Implement exception handling with TRY... CATCH ... ENDTRY. You can implement a separate CATCH

*Continued on next page*

block for each of the exception classes or use a single CATCH block for the shared superclass. If an error occurs, respond with error message 061 from message class BC402.

## Solution 13: Dynamic SELECT Statements

### Task 1:

Copy program BC402\_DYS\_DYN\_CALL (or your own program, ZBC402\_##\_DYN\_CALL) to the name **ZBC402\_##\_DYN\_SQL**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program. Analyze the source code. How is the data read from the database? Into which data objects?

**Answer:** The data is read from the database in static SELECT statements. In the process, one of two statically defined internal tables is filled.

### Task 2:

Replace the two SELECT statements with a single SELECT statement **after** the ENDCASE statement. Use the option of specifying the table name as a character-type data object. In addition, use a generically typed field symbol as the target range (INTO clause), which you dynamically assign a suitable data object.

1. Define a character-type data object for the table name (suggested name: **gv\_tabname**). Fill the data object with the respective table names in the WHEN blocks.
  - a) See the source code excerpt from the model solution.
2. Create a field symbol (suggested name: **<fs\_table>**) whose type lets it be assigned either data object gt\_conn or gt\_cust.
  - a) See the source code excerpt from the model solution.
3. Why is type ANY not suitable?

**Answer:** The field symbol should be specified after INTO **TABLE** in the SELECT statement. The syntax check only allows generic values here, with at least type ANY TABLE.
4. Assign the field symbol the respective internal tables in the WHEN blocks - the ones you want to fill in the generic SELECT statement.
  - a) See the source code excerpt from the model solution.
5. Move one of the two SELECT statements from the WHEN block to after the ENDCASE statement and delete the other one.

*Continued on next page*

Make the remaining database access dynamic: Replace the static table name with data object gv\_tabname and replace the static values of the target range with the value in the field symbol.

- See the source code excerpt from the model solution.

### Task 3: Optional:

Catch the specific runtime errors that can occur when using dynamic SQL statements.

- Read the keyword documentation to find out about the exceptions that can occur when using dynamic SQL statements. Implement exception handling with TRY... CATCH ... ENDTRY. You can implement a separate CATCH block for each of the exception classes or use a single CATCH block for the shared superclass. If an error occurs, respond with error message 061 from message class BC402.

- See the source code excerpt from the model solution.

### Result

Source code excerpt from the model solution:

```

REPORT  bc402_dys_dyn_sql MESSAGE-ID bc402.

TYPE-POOLS: abap.

DATA:
  gt_cust    TYPE ty_customers,
  gt_conn    TYPE ty_connections.

DATA:
  gv_methname TYPE string,
  gt_parmbind TYPE abap_parmbind_tab,
  gs_parmbind TYPE abap_parmbind.

DATA:
  gv_tabname  TYPE string.

FIELD-SYMBOLS:
  <fs_table> TYPE ANY TABLE.

SELECTION-SCREEN COMMENT 1(80) text-sel.
PARAMETERS:
  pa_xconn  TYPE xfeld RADIOBUTTON GROUP tab DEFAULT 'X',
  pa_xcust  TYPE xfeld RADIOBUTTON GROUP tab .

```

*Continued on next page*

```

PARAMETERS:
  pa_nol      TYPE i DEFAULT '100'.

START-OF-SELECTION.

* specific part
*-----
CASE 'X'.
  WHEN pa_xconn.

    gv_tabname = 'SPFLI'.

    ASSIGN gt_conn TO <fs_table>.

    gv_methname = 'WRITE_CONNECTIONS'.

    gs_parmbind-name = 'IT_CONN'.
    GET REFERENCE OF gt_conn INTO gs_parmbind-value.
    INSERT gs_parmbind INTO TABLE gt_parmbind.

  WHEN pa_xcust.

    gv_tabname = 'SCUSTOM'.

    ASSIGN gt_cust TO <fs_table>.

    gv_methname = 'WRITE_CUSTOMERS'.

    gs_parmbind-name = 'IT_CUST'.
    GET REFERENCE OF gt_cust INTO gs_parmbind-value.
    INSERT gs_parmbind INTO TABLE gt_parmbind.

ENDCASE.

* dynamic part
*-----
TRY.
  SELECT * FROM (gv_tabname) INTO TABLE <fs_table>
    UP TO pa_nol ROWS.
  CATCH cx_sy_dynamic_osql_error.
    MESSAGE e061.
ENDTRY.

```

*Continued on next page*

```
TRY.  
  CALL METHOD cl_bc402_utilities=>(gv_methname)  
    PARAMETER-TABLE  
      gt_parmbind.  
  
  CATCH cx_sy_dyn_call_error.  
    MESSAGE e060.  
  ENDTRY.
```



# Exercise 14: Generic Data Types and Dynamic Access to Structure Components

## Exercise Objectives

After completing this exercise, you will be able to:

- Use generically typed interface parameters
- Use generically typed field symbols
- Access structure components dynamically

## Business Example

You have been asked to develop a simple program that can display the contents of any database table. To do so, you want to find out about the options available in dynamic programming.

In the next step, you implement a generic method that lets you output the contents of any table in a list (provided the line type of the internal table has a flat structure). To do so, you replace the dynamic call of two specific methods with a static call of a generic method.

### Template:

BC402\_DYS\_DYN\_SQL (executable program)  
CL\_BC402\_DYT\_GEN\_TYPES (global class)

### Solution:

BC402\_DYS\_GEN\_TYPES (executable program)  
CL\_BC402\_DYS\_GEN\_TYPES (global class)

### Task 1:

Copy global class CL\_BC402\_DYT\_GEN\_TYPES to the name  
**ZCL\_BC402\_##\_GEN\_TYPES**, where ## is your group number. Familiarize yourself with the class signature.

1. Copy the global class.

*Continued on next page*

2. Analyze the class signature. Which methods are defined? What is their purpose?

---

---

---

3. Which parameters do these methods have? How are the parameters typed?

---

---

---

### Task 2:

First implement method write\_any\_struct. Output all components of structure gs\_struct individually with a WRITE statement. To do so, program a loop in which you access all the components in structure gs\_struct consecutively.

1. Create a field symbol (suggested name: <fs\_comp>) that has a generic type that lets it be assigned any flat components of the structure.
2. Program a DO loop. Use the sy-index loop counter to access all components of gs\_struct consecutively with the field symbol.
3. Check whether the field symbol assignment was successful. If it was, output the component with a WRITE. Otherwise, exit the loop.

### Task 3:

Now implement method write\_any\_table. Display the contents of internal table it\_table in a list by lines. To do so, program a loop over internal table gt\_table and call up your previously implemented method write\_any\_struct for each line.

1. Create a field symbol (suggested name: <fs\_line>) that has a generic type that lets it be assigned a line of generic internal table it\_table.
2. Program a LOOP over internal table it\_table. Assign the lines in the internal table to the field symbol, <fs\_line>, consecutively.
3. Call method write\_any\_struct for each line.

*Continued on next page*

## Task 4:

Copy program BC402\_DYS\_DYN\_SQL (or your own program, ZBC402##\_DYN\_SQL) to the name **ZBC402##\_GEN\_TYPES**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
2. Activate and test the program. Analyze the source code.

## Task 5:

Replace the dynamic method call at the end of the program with a call of your new method, write\_any\_table. Remove (or flag as comments) all parts of the source code you no longer need.

1. Delete the dynamic method call or flat it as a comment. In its place, implement a call of method write\_any\_table. Give the method the generic field symbol that points to the internal table with the data.
2. Delete the definition of the data objects that are needed for the dynamic method call or flag them as comments.
3. Activate and test the program.

## Solution 14: Generic Data Types and Dynamic Access to Structure Components

### Task 1:

Copy global class CL\_BC402\_DYT\_GEN\_TYPES to the name **ZCL\_BC402\_##\_GEN\_TYPES**, where ## is your group number. Familiarize yourself with the class signature.

1. Copy the global class.
  - a) Carry out this step in the usual manner.
2. Analyze the class signature. Which methods are defined? What is their purpose?  
**Answer:** write\_any\_struct and write\_any\_table for list output of any flat structure or any internal table with flat line type.
3. Which parameters do these methods have? How are the parameters typed?  
**Answer:** gs\_struct TYPE ANY and gt\_struct TYPE ANY TABLE

### Task 2:

First implement method write\_any\_struct. Output all components of structure gs\_struct individually with a WRITE statement. To do so, program a loop in which you access all the components in structure gs\_struct consecutively.

1. Create a field symbol (suggested name: **<fs\_comp>**) that has a generic type that lets it be assigned any flat components of the structure.
  - a) See the source code excerpt from the model solution.
2. Program a DO loop. Use the sy-index loop counter to access all components of gs\_struct consecutively with the field symbol.
  - a) See the source code excerpt from the model solution.
3. Check whether the field symbol assignment was successful. If it was, output the component with a WRITE. Otherwise, exit the loop.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

### Task 3:

Now implement method write\_any\_table. Display the contents of internal table it\_table in a list by lines. To do so, program a loop over internal table gt\_table and call up your previously implemented method write\_any\_struct for each line.

1. Create a field symbol (suggested name: <fs\_line>) that has a generic type that lets it be assigned a line of generic internal table it\_table.
  - a) See the source code excerpt from the model solution.
2. Program a LOOP over internal table it\_table. Assign the lines in the internal table to the field symbol, <fs\_line>, consecutively.
  - a) See the source code excerpt from the model solution.
3. Call method write\_any\_struct for each line.
  - a) See the source code excerpt from the model solution.

### Task 4:

Copy program BC402\_DYS\_DYN\_SQL (or your own program, ZBC402\_##\_DYN\_SQL) to the name **ZBC402\_##\_GEN\_TYPES**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program. Analyze the source code.
  - a) Carry out this step in the usual manner.

### Task 5:

Replace the dynamic method call at the end of the program with a call of your new method, write\_any\_table. Remove (or flag as comments) all parts of the source code you no longer need.

1. Delete the dynamic method call or flat it as a comment. In its place, implement a call of method write\_any\_table. Give the method the generic field symbol that points to the internal table with the data.
  - a) See the source code excerpt from the model solution.
2. Delete the definition of the data objects that are needed for the dynamic method call or flag them as comments.
  - a) See the source code excerpt from the model solution.
3. Activate and test the program.
  - a) Carry out this step in the usual manner.

*Continued on next page*

## Result

Source code excerpt from the model solution:

```
Method write_any_struct (Class
CL_BC402_DYS_GEN_TYPE)

method WRITE_ANY_STRUCT.

FIELD-SYMBOLS <fs_comp> TYPE simple.

NEW-LINE.
DO.
  ASSIGN COMPONENT sy-index OF STRUCTURE is_struct TO <fs_comp>.
  IF sy-subrc = 0.
    WRITE <fs_comp>.
  ELSE.
    EXIT.
  ENDIF.
ENDDO.

endmethod.
```

```
Method write_any_table (Class
CL_BC402_DYS_GEN_TYPE)
```

```
method WRITE_ANY_TABLE.

FIELD-SYMBOLS: <fs_line> TYPE ANY.

LOOP AT it_table ASSIGNING <fs_line>.
  write_any_struct( <fs_line> ).
ENDLOOP.

endmethod.
```

## Executable Program BC402\_DYS\_GEN\_TYPES

```
REPORT bc402_dys_gen_types MESSAGE-ID bc402.
```

*Continued on next page*

```

*TYPE-POOLS: abap.

DATA:
  gt_cust      TYPE ty_customers,
  gt_conn      TYPE ty_connections.

*DATA:
*  gv_methname TYPE string,
*  gt_parmbind TYPE abap_parmbind_tab,
*  gs_parmbind TYPE abap_parmbind.

DATA:
  gv_tabname  TYPE string.

FIELD-SYMBOLS:
  <fs_table> TYPE ANY TABLE.

SELECTION-SCREEN COMMENT 1(80) text-sel.
PARAMETERS:
  pa_xconn  TYPE xfeld RADIobutton GROUP tab DEFAULT 'X',
  pa_xcust  TYPE xfeld RADIobutton GROUP tab .
PARAMETERS:
  pa_nol    TYPE i DEFAULT '100'.

START-OF-SELECTION.

* specific part
*-----
CASE 'X'.
  WHEN pa_xconn.

    gv_tabname = 'SPFLI'.
    ASSIGN gt_conn TO <fs_table>.

  WHEN pa_xcust.

    gv_tabname = 'SCUSTOM'.
    ASSIGN gt_cust TO <fs_table>.

ENDCASE.

* dynamic part

```

*Continued on next page*

```
*-----*  
  
TRY.  
  SELECT * FROM (gv_tabname) INTO TABLE <fs_table>  
    UP TO pa_nol ROWS.  
  CATCH cx_sy_dynamic_osql_error.  
    MESSAGE e061.  
  ENDTRY.  
  
* TRY.  
*   CALL METHOD cl_bc402_utilities=>(gv_methname)  
*     PARAMETER-TABLE  
*       gt_parmbind.  
*  
*   CATCH cx_sy_dyn_call_error.  
*     MESSAGE e060.  
* ENDTRY.  
  
cl_bc402_dys_gen_types=>write_any_table( <fs_table> ).
```



## Lesson Summary

You should now be able to:

- Explain what generic types are and what they are used for
- Use generically typed parameters, field symbols, and data references
- Use field symbols to access data objects dynamically
- Use field symbols to access attributes and structure components dynamically

# Lesson: Querying Type Attributes at Runtime

## Lesson Overview

In this lesson, you learn about runtime type identification - a technique that lets you query information about data types and object types at runtime. This technique is essential for using generically typed parameters, field symbols, and references in dynamic programs.



## Lesson Objectives

After completing this lesson, you will be able to:

- Query the properties of data objects and data types at runtime
- Query the properties of classes and instances at runtime

## Business Example

You have been asked to develop a flexible application. To do so, you use generically typed parameters, field symbols, and references. You want to learn about the options provided by runtime type identification for querying information about the actual type of the underlying (data) object at runtime.

## Runtime Type Identification

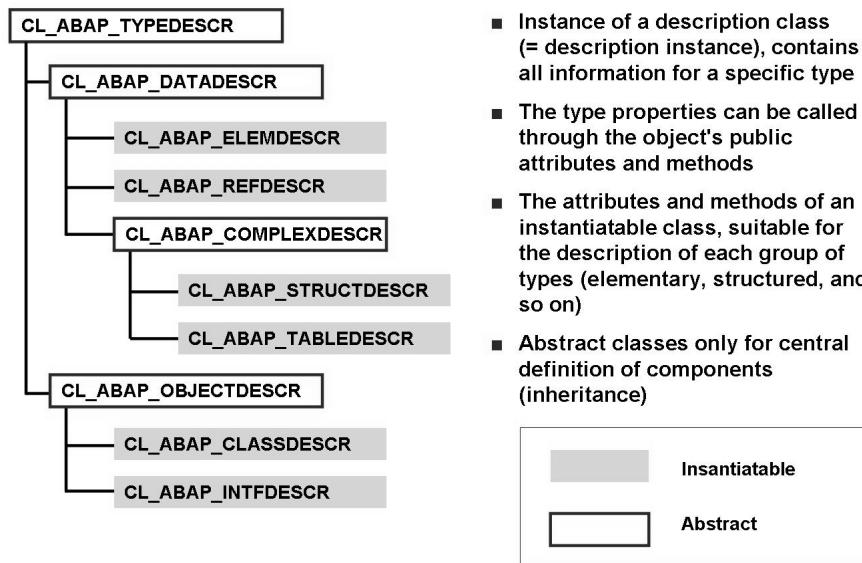
As of *SAP Web AS 6.10*, ABAP developers can use a class-based concept to determine type properties (data types and object types) at runtime. This concept is called runtime type identification.



**Hint:** Before runtime type identification, ABAP only had the DESCRIBE FIELD and DESCRIBE TABLE statements. These statements are limited to properties of data objects, however, and cannot determine as many properties as runtime type identification.

Together with the lines() function – which was also introduced in *SAP Web AS 6.10* – runtime type identification makes the DESCRIBE FIELD and DESCRIBE TABLE obsolete.

Runtime type identification consists of a hierarchy of 10 global classes that developers can use:



**Figure 157: Class Hierarchy of RTTI Description Classes**

The description of a type at runtime is nothing more than an instance of one of these classes. The properties of the type are saved in the instance attributes and can be queried directly or using appropriate methods. At runtime, only one description object exists for each type.

All classes inherit properties from a shared superclass, either directly or indirectly, and their names all follow the pattern **CL\_ABAP\_xxxDESCR**, where **xxx** stands for the category of type used to describe the respective class.

Different classes are needed because the attributes and methods of each class specialize it to describe a specific category of types. Class **CL\_ABAP\_TABLEDESCR**, for example, is used to describe table types. Therefore, it has attributes for the table category and the structure of the table key, among other information. These attributes are specific to class **CL\_ABAP\_TABLEDESCR** and are missing in all other RTTI classes.

Only six of the ten RTTI classes can be instantiated and used to describe specific types. They are:



### **CL\_ABAP\_ELEMDESCR**

To describe elementary data types

### **CL\_ABAP\_REFDESCR**

To describe reference types (=types of reference variables)

### **CL\_ABAP\_STRUCTDESCR**

To describe structure types

**CL\_ABAP\_TABLEDESCR**

To describe table types

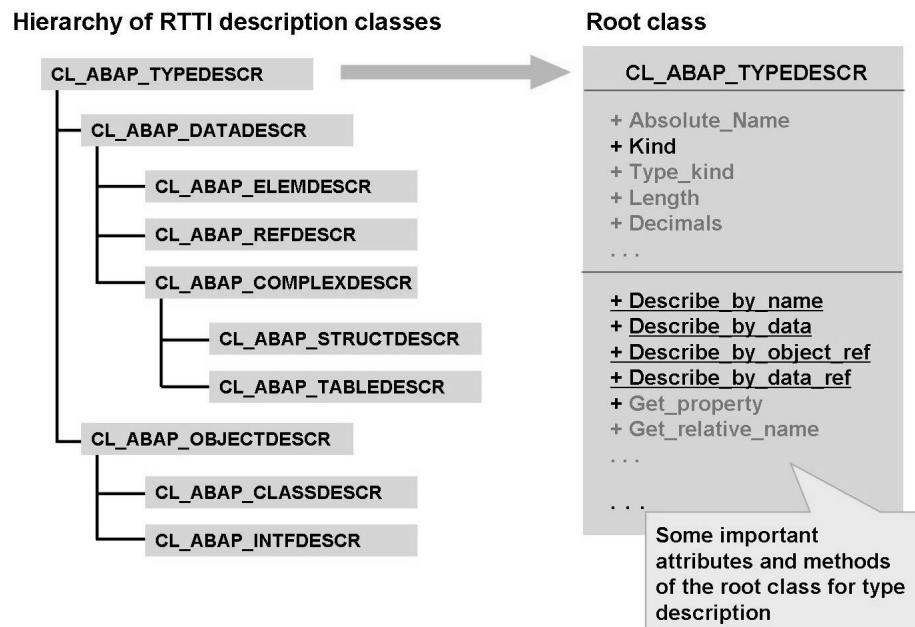
**CL\_ABAP\_CLASSDESCR**

To describe classes (=object types)

**CL\_ABAP\_INTFDESCR**

To describe interfaces

All other classes are abstract, which means they cannot be instantiated. They are used to centrally define the attributes and methods that are used in several of the other classes (and implement them if necessary). The METHODS attribute, for example, which contains a list of the methods, is not defined in class CL\_ABAP\_CLASSDESCR, but instead in class CL\_ABAP\_OBJECTDESCR, as it is also needed in the same form in class CL\_ABAP\_INTFDESCR.



**Figure 158: RTTI - Methods and Attributes of the Root Class**

The RTTI description classes cannot be instantiated directly with the CREATE OBJECT statement. To retrieve a reference to a description object, you have to call static method DESCRIBE\_BY\_xxx of class CL\_ABAP\_TYPEDESCR. The description objects are generated and the data is filled and returned using a return parameter with generic type REF TO CL\_ABAP\_TYPEDESCR.



```
DATA lo_type TYPE REF TO cl_abap_typedescr.
```

Analysis of a local data type

```
TYPES lty_type TYPE ....  
lo_type = cl_abap_typedescr=>describe_by_name( 'LTY_TYPE' ).
```

Analysis of a global data type

```
lo_type = cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
```

Analysis of a local object type (for example, local class)

```
CLASS lcl_class DEFINITION.  
...  
ENDCLASS.  
...  
lo_type = cl_abap_typedescr=>describe_by_name( 'LCL_CLASS' ).
```

Analysis of a global object type (for example, global interface)

```
lo_type = cl_abap_typedescr=>describe_by_name( 'IF_PARTNERS' ).
```

**Figure 159: Describing a Type Based on Its Name**

After the call, you have to cast a suitable subclass, to access the specific attributes and methods for the respective type. If you do not know which RTTI class was actually instantiated, you can use the public KIND instance attribute. Its contents match the value of one of six constants from class CL\_ABAP\_TYPEDESCR. Each of these six values corresponds to one of the six RTTI classes that can be instantiated.



```
DATA:lo_type      TYPE REF TO cl_abap_typedescr,
      lo_elem     TYPE REF TO cl_abap_elemdescr,
      lo_ref      TYPE REF TO cl_abap_refdescr,
      lo_struct   TYPE REF TO cl_abap_structdescr,
      lo_table    TYPE REF TO cl_abap_tabledescr,
      lo_intf     TYPE REF TO cl_abap_intfdescr,
      lo_class    TYPE REF TO cl_abap_classdescr.
```

```
lo_type = cl_abap_typedescr=>describe_by_ ... .

CASE lo_type->kind.
  WHEN cl_abap_typedescr=>kind_elem.
    lo_elem ?= lo_type.
  WHEN cl_abap_typedescr=>kind_ref.
    lo_ref ?= lo_type.
  WHEN cl_abap_typedescr=>kind_struct.
    lo_struct ?= lo_type.
  WHEN cl_abap_typedescr=>kind_table.
    lo_table ?= lo_type.
  WHEN cl_abap_typedescr=>kind_intf.
    lo_intf ?= lo_type.
  WHEN cl_abap_typedescr=>kind_class.
    lo_class ?= lo_type.
ENDCASE.
```

Evaluation of KIND  
(child) attribute, then  
down cast to correct  
subclass

**Figure 160: Casing a Suitable Reference for a Type Description Object**



**Hint:** If you already know which description class is instantiated (because a class name is always passed on, for example), then the down cast can also take place directly in the call of the functional method - for example:

```
lo_class ?= cl_abap_typedescr=>describe_by_ ...
```

In this case, it can never hurt to catch runtime error  
`CX_SY_MOVE_CAST_ERROR`.

The four methods differ with regard to which information they must be given in their calls:

#### **DESCRIBE\_BY\_NAME**

Returns the description object whose type name is known. Can analyze both data types and object types (classes and interfaces). It does not matter whether the type was defined globally in the Dictionary or Class Builder or locally in the program.

The name must be in all caps. If no type with the specified name is found, the method raises the (conventional) exception `TYPE_NOT_FOUND`.

### **DESCRIBE\_BY\_DATA**

Returns the description object for the type of a data object. This method only returns descriptions of data types. It does not matter whether the data object was defined with an explicit type or a bound type.

If the specified data object is a parameter or field symbol with a generic type, the description of the generic type is not returned, but instead the type of the currently assigned actual parameter or data object.

### **DESCRIBE\_BY\_DATA\_REF**

Returns the description object for the type of a data object that points to a data reference. This method only returns descriptions of data types. If the data reference is typed generically (TYPE REF TO DATA), this method can be used to determine the dynamic type of the referenced object.



**Caution:** If the reference does not have a valid value at call time, the method raises the (conventional) exception `REFERENCE_IS_INITIAL`.

### **DESCRIBE\_BY\_OBJECT\_REF**

Returns the description object for the type of an instance that points to an object reference. This method only returns descriptions of classes. You can use this method to determine the dynamic type of the referenced instance after an up cast to a superclass or implemented interface.



**Caution:** If the reference does not have a valid value at call time, the method raises the (conventional) exception `REFERENCE_IS_INITIAL`.

The diagram below shows examples of how to use these methods:



Analysis of a generic parameter

```
... IMPORTING ig_data_object TYPE any ...
...
lo_type = cl_abap_typedescr->describe_by_data( ig_data_object ).
```

Returns description of type  
of current parameter

Analysis of a reference variable (object reference)

```
DATA: lo_vehicle TYPE REF TO lcl_vehicle.
...
lo_type = cl_abap_typedescr->describe_by_data( lo_vehicle ).
```

Returns type description  
of reference variable

```
lo_type = cl_abap_typedescr->describe_by_object_ref( lo_vehicle ).
```

Returns type description of  
object, reference variable is  
pointing to

**Figure 161: Describing Types Based on Data Objects and References**

In addition to the static methods listed above, another option is available for accessing description objects. You use special methods to navigate from one description object to other description objects. Example: You want to navigate from the description of a table type to the description of the corresponding line type. Or you want to navigate from a structure type to the description objects for the component types. We discuss these options based on examples below.

## Dynamic Data Type Analysis

In the following application examples, you see how you can use RTTI to determine the properties of different data types. Whether or not a given subclass is used depends on the data type.

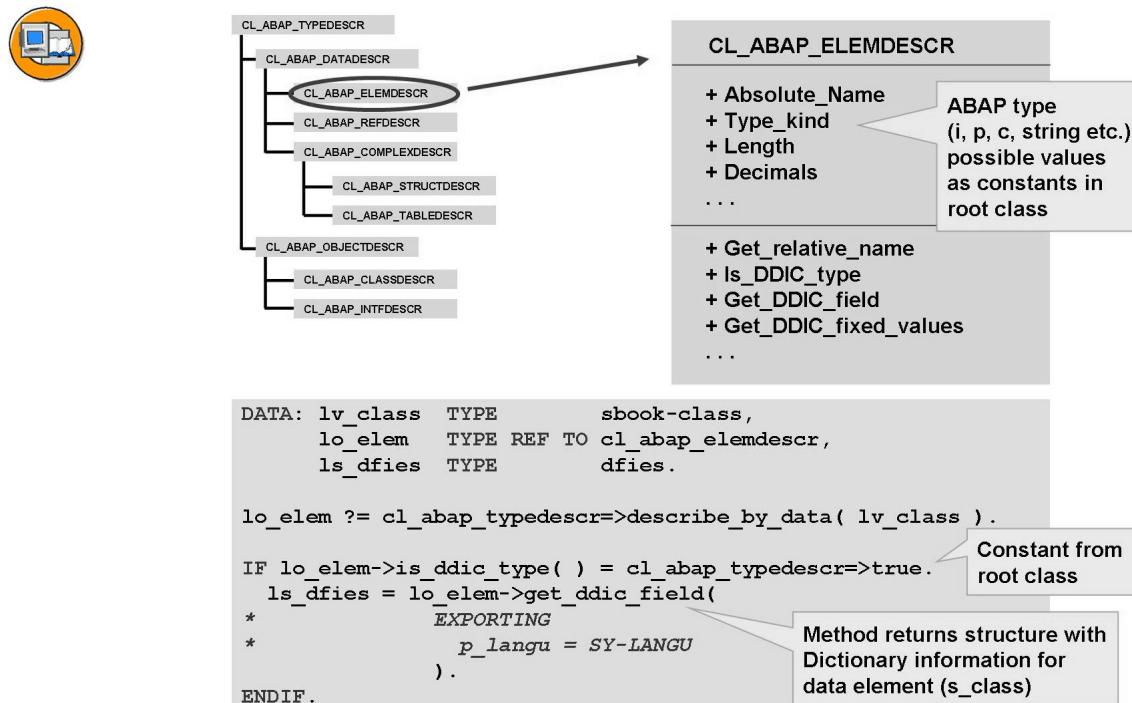


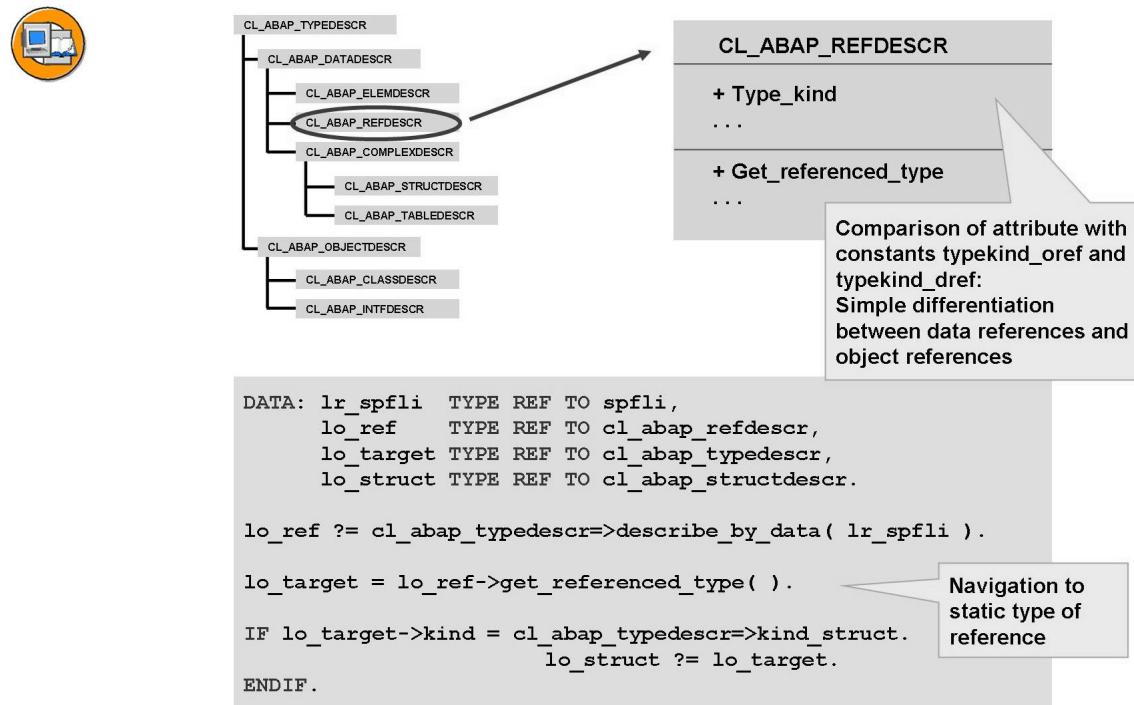
Figure 162: Analysis of an Elementary Data Type

An instance of class **CL\_ABAP\_ELEMDESCR** describes all the properties of an elementary data type. The **technical properties** of the underlying ABAP type, the length, and - if relevant - the number of decimal places are located in the corresponding **public attributes**. To evaluate the **TYPE\_KIND** attribute, you must compare its contents with the corresponding constants from class **CL\_ABAP\_TYPEDESCR**, similar to the **KIND** attribute.

→ **Note:** Note that the three attributes are described in the root class of RTTI, which means they are inherited by all description classes. These attributes are less (or not at all) significant in the other classes, however.

The **semantic properties** - that is, the additional information from the *ABAP Dictionary* - must be determined explicitly using public methods. The above example shows a call of method `GET_DDIC_FIELD`. An optional parameter lets you determine the language-dependent information (field labels, and so on) in a language other than the current logon language.

⚠ **Caution:** If the current type is not a Dictionary type, the method raises a (conventional) exception. If you want to avoid this, you can use method `IS_DDIC_TYPE` to check in advance whether Dictionary information is available (see above example).



**Figure 163: Analysis of the Type of a Reference Variable**

When you analyze a reference type (=type of a reference variable), the inherited TYPE\_KIND attribute lets you easily differentiate between object references and data references. To determine the details of the static type of the reference variable, however, you have to use the GET\_REFERENCED\_TYPE navigation method. It returns a reference to another description object. This description object can be an instance of class CL\_ABAP\_STRUCTDESCR - as in the above example - if the reference variable was created with a structure type. If the reference variable was created with reference to a class or an interface, however, then GET\_REFERENCED\_TYPE returns instances of the CL\_ABAP\_CLASSDESCR or CL\_ABAP\_INTFDESCR class, respectively.

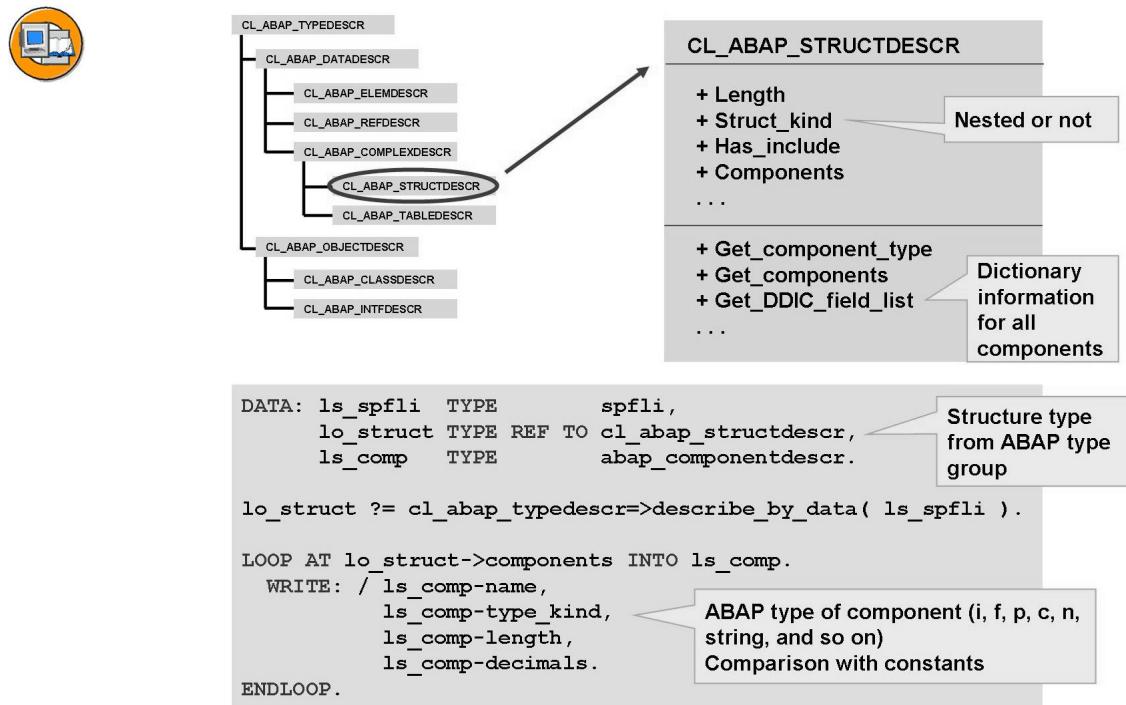


Figure 164: Analysis of a Structure Type

The CL\_ABAP\_STRUCTDESCR class also has a number of public attributes with basic information, such as the total length of bytes, the existence of includes, the type of structure (nested or not), and so on. The COMPONENTS attribute is an internal table that contains the names of all components and their elementary technical properties.

You can use the GET\_DDIC\_FIELD\_LIST method to determine the semantic information (if any) for the components. It is similar to the GET\_DDIC\_FIELD method, but returns an internal table instead of a structure.



Type description of a specific component

```
DATA: lo_struct TYPE REF TO cl_abap_structdescr,
      lo_comp   TYPE REF TO cl_abap_datadescr.

* fill lo_struct with reference to structure type

lo_comp = lo_struct->get_component_type( 'CARRID' ).
```

Name of component  
as literal or variable

Type description of all components

```
DATA: lo_struct TYPE REF TO cl_abap_structdescr,
      lo_elem   TYPE REF TO cl_abap_elemdescr,
      lt_comp   TYPE cl_abap_structdescr=>component_table,
      ls_comp   TYPE cl_abap_structdescr=>component.

* fill lo_struct with reference to structure type

lt_comp = lo_struct->get_components( ).

LOOP AT lt_comp INTO ls_comp.
  IF ls_comp-type->kind = cl_abap_typedescr=>kind_elem.
    lo_elem ?= ls_comp-type.
    ...
  ENDIF.
ENDLOOP.
```

Types are defined  
in the class

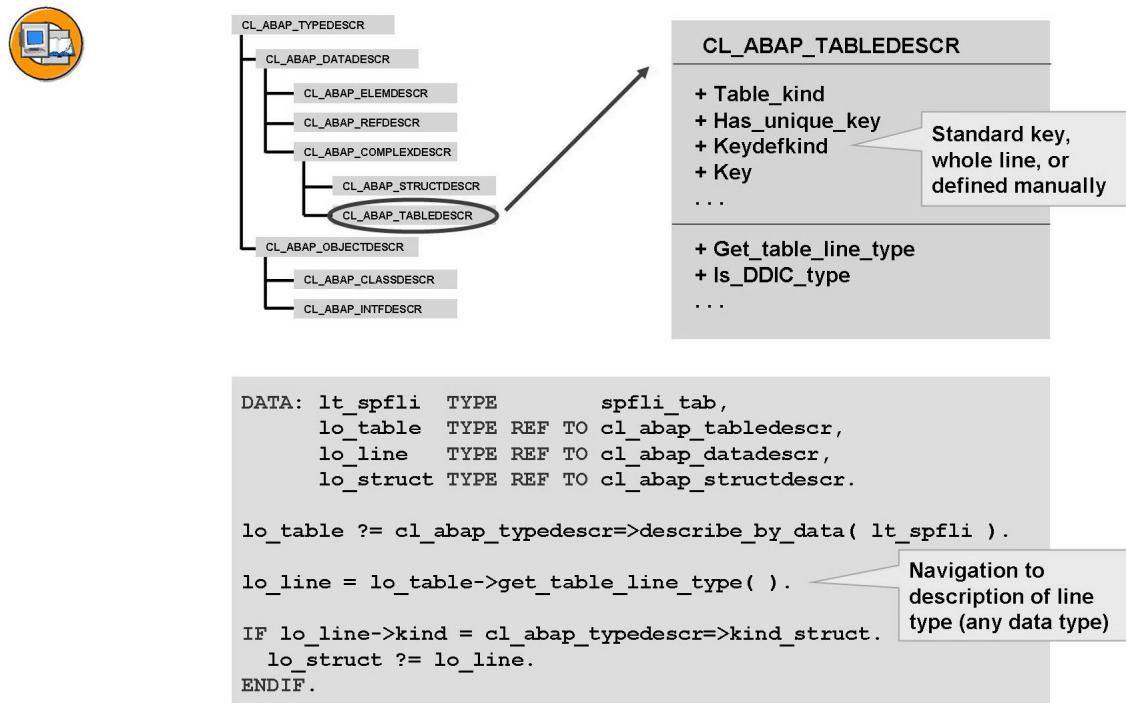
Table with component names  
and references to corresponding  
description objects

**Figure 165: Navigation from Structure Type to Component Types**

To gain full access to all properties of the component types, you have to navigate to the respective description objects. The GET\_COMPONENT\_TYPE navigation method returns the description object for the type of a specific component, while the GET\_COMPONENTS navigation method returns the description objects for all components.



**Hint:** Since component types can also be reference types, structure types, or table types, the return values are defined with type REF TO CL\_ABAP\_DATADESCRIPTOR. A down cast to one of the subclasses may be necessary in reference variables, for example, to type REF TO CL\_ABAP\_ELEMDESCR.



**Figure 166: Analysis of a Table Type**

The public attributes of class CL\_ABAP\_TABLEDESCR include the table type (standard, sorted, hashed, index, any), type and uniqueness of the key, and a list of names of the key components.

To determine the details for a line type, you have to navigate to a description object for that line type. To do so, use the GET\_TABLE\_LINE\_TYPE navigation method.



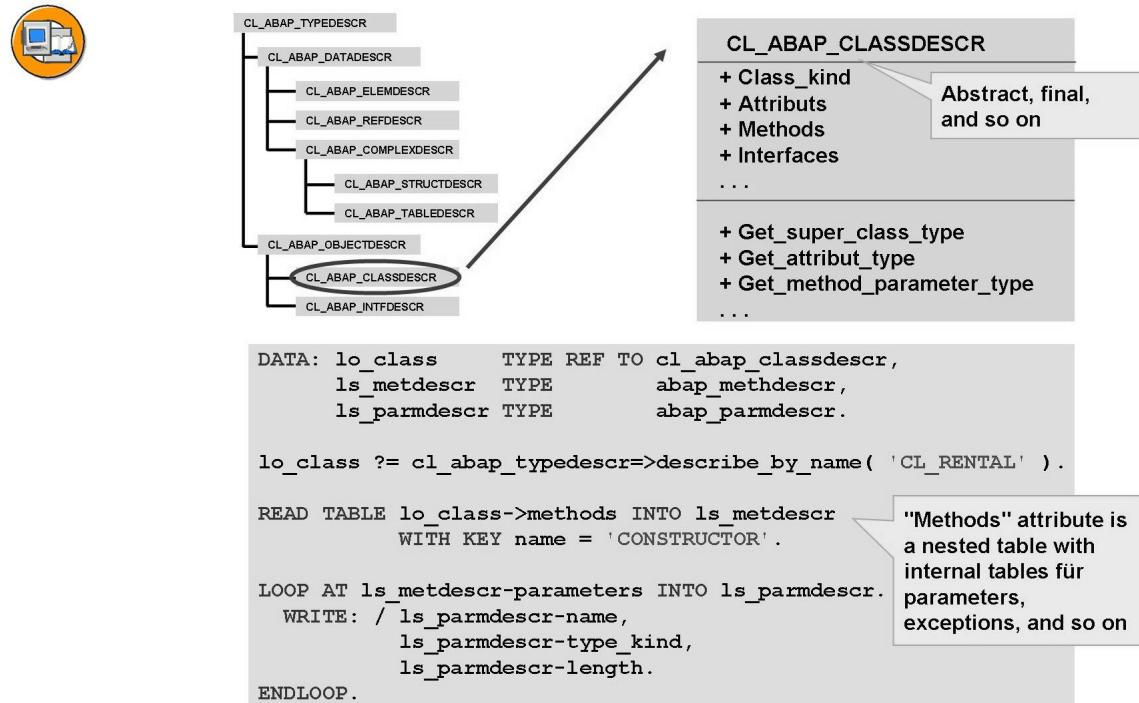
**Hint:** The line type of a table type can have any data type. Accordingly, the return parameter is typed with REF TO CL\_ABAP\_DATADESCR, and you may have to perform a down cast to a suitable type.

## Dynamic Object Type Analysis

RTTI also permits you to analyze the attributes of objects. Classes CL\_ABAP\_CLASSDESCR and CL\_ABAP\_INTFDESCR are available to do so.

Most of the attributes and methods for these two classes are defined in the (abstract) shared superclass CL\_ABAP\_OBJECTDESCR, because they are needed to analyze classes and interfaces in equal measure.

→ **Note:** A class is analyzed in the example below. Most of the information also applies to the analysis of interfaces.



**Figure 167: Analysis of an Object Type**

Class CL\_ABAP\_CLASSDESCR contains public attributes in which the attributes, methods, events, interfaces, and other objects of the described class are listed. These attributes are typed as internal tables. Some of these attributes – such as METHODS – are defined as nested internal tables, which means each line of the internal table itself contains one more internal tables. These inner internal tables can contain lists with parameters and exceptions for the respective method, for example.

In addition to these inherited attributes, class CL\_ABAP\_CLASSDESCR also provides specific attributes – such as the CLASS\_KIND attribute, which you can use to determine whether the class was flagged as abstract, final, and so on.

→ **Note:** Class CL\_ABAP\_INTFDESCR contains an INTF\_KIND attribute instead, which lets you determine whether or not an interface is comprised of multiple interfaces.

The above example shows how RTTI is used to determine all the parameters of the constructor for class CL\_RENTAL.

After the description object is generated and the down cast is performed, the description of the constructor method is read first from the list of methods (statement with READ TABLE). The list of parameters for this method is then evaluated in a loop (LOOP over the inner internal table).

The description classes for object types also provide **navigation methods** for determining the details of a used type. The GET\_ATTRIBUTE\_TYPE method, for example, returns the description object for the type of a specific attribute, while GET\_METHOD\_PARAMETER\_TYPE analyzes the type of a specific method parameter, and so on.

The GET\_SUPER\_CLASS\_TYPE method is specific to classes. It returns the description object for the direct superclass (if any).



## Exercise 15: Optional: Column Headers with Runtime Type Identification

### Exercise Objectives

After completing this exercise, you will be able to:

- Use RTTI to analyze a structure type
- Use RTTI to determine semantic information for a structure type from the *ABAP Dictionary*
- Use the semantic information for a structure type to generate column headers dynamically

### Business Example

You have been asked to develop a simple program that can display the contents of any database table. To do so, you want to find out about the options available in dynamic programming.

You have already developed a method to output any internal table with a flat line type to a list. In the next step, you want to develop a method that generates dynamic column headers for it. To do so, you use the options of runtime type identification (RTTI).

#### Template:

BC402\_DYS\_GEN\_TYPES (executable program)  
CL\_BC402\_DYT\_RTTI\_HEADERS (global class)

#### Solution:

BC402\_DYS\_RTTI\_HEADERS (executable program)  
CL\_BC402\_DYS\_RTTI\_HEADERS (global class)

#### Task 1:

Copy the template CL\_BC402\_DYT\_RTTI\_HEADERS to the name **ZCL\_BC402\_##\_RTTI\_HEADERS**, where ## is your group number. Familiarize yourself with the class signature.

1. Copy the global class.

*Continued on next page*

2. Analyze the class signature. Which methods are defined? What is their purpose?

---

---

---

3. Which parameters and exceptions do these methods have? How are the parameters typed? Which error situations correspond to the exceptions?

---

---

---

## Task 2:

Implement method write\_headers. Use the options of runtime type identification (RTTI) to analyze the type whose name is contained in import parameter gv\_tabname.

1. Call a suitable static method of class CL\_ABAP\_TYPEDESCR to analyze the data type based on its name.
2. Create a local reference variable (suggested name: **lo\_type**) and type it to match the return parameter of the method. Specify this reference variable as an actual parameter in the method call.
3. Handle possible exceptions and raise exception TYPE\_NOT\_FOUND from method write\_headers as a reaction.
4. Use the public KIND attribute to find out whether the generated RTTI instance actually describes a structure type. If the data type is not a structure type, exit method write\_headers with exception NO\_STRUCTURE.
5. Create a local reference variable (suggested name: **lo\_struct**) and assign it a type that lets it point to instances of class CL\_ABAP\_STRUCTDESCR. Program a down cast to point this reference to the type description object.



**Hint:** You do not have to catch possible runtime errors here, as the query of the KIND attribute already ensured compatibility.

*Continued on next page*

### Task 3:

Determine a list with Dictionary information for all components of the structure type. Output the report header for all components in sequence. Make sure the output does not exceed the display length of the corresponding field.

1. Create an internal table with type ddfIELDS (suggested name: **lt\_fields**).
2. Call a suitable method of class CL\_ABAP\_STRUCTDESCR to generate a list with Dictionary information for the structure components. To do so, create an internal table with type ddfIELDS as the actual parameter for the return value (suggested name: **lt\_fields**). If the method call terminates with exceptions, exit the write\_headers method with a suitable exception.
3. Program a loop over the internal table. To do so, create a field symbol (suggested name: **<fs\_flist>**) that is typed with the line type of internal table **lt\_fields**.
4. Output the report header (field *reptext*) in each loop pass. Use the output length (field *outputlen*) for length access, to ensure the headers are exactly as long as they will be output later.

### Task 4:

Copy program BC402\_DYS\_GEN\_TYPES (or your own program, ZBC402\_##\_GEN\_TYPES) to the name **ZBC402\_##\_RTTI\_HEADERS**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
2. Activate and test the program. Analyze the source code.

### Task 5:

Insert a call of your new method, `write_headers`, before the data output. Pass on the name of the line type of the internal table with the data to your method.

1. Implement a call of method `write-headers` before the call of method `write_any_table`. Pass on the data object that contains the name of the database table (and therefore the name of the line type of the internal table).
2. React to exceptions with suitable error messages from message class BC402.
3. Activate and test the program.

## Solution 15: Optional: Column Headers with Runtime Type Identification

### Task 1:

Copy the template CL\_BC402\_DYT\_RTTI\_HEADERS to the name **ZCL\_BC402\_##\_RTTI\_HEADERS**, where ## is your group number. Familiarize yourself with the class signature.

1. Copy the global class.
  - a) Carry out this step in the usual manner.
2. Analyze the class signature. Which methods are defined? What is their purpose?

**Answer:** write\_headers to output a list of column headers based on a structure type from the *ABAP Dictionary*.

3. Which parameters and exceptions do these methods have? How are the parameters typed? Which error situations correspond to the exceptions?

**Answer:** Parameter iv\_tabname with generic type csequence and exceptions

#### **TYPE\_NOT\_FOUND**

The passed on type is unknown

#### **NO\_STRUCTURE**

The object is not a structure type

#### **NO\_DDIC\_TYPE**

The object is not a type in the ABAP Dictionary

### Task 2:

Implement method write\_headers. Use the options of runtime type identification (RTTI) to analyze the type whose name is contained in import parameter gv\_tabname.

1. Call a suitable static method of class CL\_ABAP\_TYPEDESCR to analyze the data type based on its name.
  - a) See the source code excerpt from the model solution.
2. Create a local reference variable (suggested name: **lo\_type**) and type it to match the return parameter of the method. Specify this reference variable as an actual parameter in the method call.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

3. Handle possible exceptions and raise exception TYPE\_NOT\_FOUND from method write\_headers as a reaction.
  - a)
4. Use the public KIND attribute to find out whether the generated RTTI instance actually describes a structure type. If the data type is not a structure type, exit method write\_headers with exception NO\_STRUCTURE.
  - a) See the source code excerpt from the model solution.
5. Create a local reference variable (suggested name: **lo\_struct**) and assign it a type that lets it point to instances of class CL\_ABAP\_STRUCTDESCR. Program a down cast to point this reference to the type description object.



**Hint:** You do not have to catch possible runtime errors here, as the query of the KIND attribute already ensured compatibility.

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

### Task 3:

Determine a list with Dictionary information for all components of the structure type. Output the report header for all components in sequence. Make sure the output does not exceed the display length of the corresponding field.

1. Create an internal table with type ddfIELDS (suggested name: **lt\_fields**).
  - a) See the source code excerpt from the model solution.
2. Call a suitable method of class CL\_ABAP\_STRUCTDESCR to generate a list with Dictionary information for the structure components. To do so, create an internal table with type ddfIELDS as the actual parameter for the return value (suggested name: **lt\_fields**). If the method call terminates with exceptions, exit the write\_headers method with a suitable exception.
  - a) See the source code excerpt from the model solution.
3. Program a loop over the internal table. To do so, create a field symbol (suggested name: **<fs\_flist>**) that is typed with the line type of internal table lt\_fields.
  - a) See the source code excerpt from the model solution.
4. Output the report header (field *reptext*) in each loop pass. Use the output length (field *outputlen*) for length access, to ensure the headers are exactly as long as they will be output later.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

## Task 4:

Copy program BC402\_DYS\_GEN\_TYPES (or your own program, ZBC402\_##\_GEN\_TYPES) to the name **ZBC402\_##\_RTTI\_HEADERS**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program. Analyze the source code.
  - a) Carry out this step in the usual manner.

## Task 5:

Insert a call of your new method, write\_headers, before the data output. Pass on the name of the line type of the internal table with the data to your method.

1. Implement a call of method write-headers before the call of method write\_any\_table. Pass on the data object that contains the name of the database table (and therefore the name of the line type of the internal table).
  - a) See the source code excerpt from the model solution.
2. React to exceptions with suitable error messages from message class BC402.
  - a) See the source code excerpt from the model solution.
3. Activate and test the program.
  - a) Carry out this step in the usual manner.

## Result

Source code excerpt from the model solution:

### **Method write\_headers (Class CL\_BC402\_DYS\_RTTI\_HEADERS)**

```
METHOD write_headers.

DATA:
  lo_type    TYPE REF TO cl_abap_typedescr,
  lo_struct  TYPE REF TO cl_abap_structdescr,
  lt_flist   TYPE ddfIELDS.

FIELD-SYMBOLS:
  <fs_flist> LIKE LINE OF lt_flist.
```

*Continued on next page*

```

CALL METHOD cl_abap_typedescr=>describe_by_name
      EXPORTING
        p_name          = iv_tabname
      RECEIVING
        p_descr_ref    = lo_type
      EXCEPTIONS
        type_not_found = 1.
      IF sy-subrc <> 0.
        RAISE type_not_found.
      ENDIF.

      IF lo_type->kind <> cl_abap_typedescr->kind_struct.
        RAISE no_structure.
      ENDIF.

      lo_struct ?= lo_type.

      CALL METHOD lo_struct->get_ddic_field_list
            RECEIVING
              p_field_list      = lt_flist
            EXCEPTIONS
              not_found          = 1
              no_ddic_type       = 2.

      CASE sy-subrc.
        WHEN 1.
          RAISE type_not_found.
        WHEN 2.
          RAISE no_ddic_type.
      ENDCASE.

      NEW-LINE.

      LOOP AT lt_flist ASSIGNING <fs_flist>.
        WRITE <fs_flist>-reptext(<fs_flist>-outputlen) COLOR COL_HEADING.
      ENDLOOP.

      ULINE.

ENDMETHOD.

```

*Continued on next page*

**Executable Program BC402\_DYS\_RTTI\_HEADERS**

```

REPORT  bc402_dys_rtti_headers MESSAGE-ID bc402.

TYPE-POOLS: abap.

DATA:
  gt_cust      TYPE ty_customers,
  gt_conn      TYPE ty_connections.

DATA:
  gv_tabname   TYPE string.

FIELD-SYMBOLS:
  <fs_table>  TYPE ANY TABLE.

SELECTION-SCREEN COMMENT 1(80) text-sel.
PARAMETERS:
  pa_xconn    TYPE xfeld RADIobutton GROUP tab DEFAULT 'X',
  pa_xcust    TYPE xfeld RADIobutton GROUP tab .
PARAMETERS:
  pa_nol      TYPE i DEFAULT '100'.

START-OF-SELECTION.

* specific part
*-----
CASE 'X'.
  WHEN pa_xconn.

    gv_tabname = 'SPFLI'.
    ASSIGN gt_conn TO <fs_table>.

  WHEN pa_xcust.

    gv_tabname = 'SCUSTOM'.
    ASSIGN gt_cust TO <fs_table>.

ENDCASE.

* dynamic part
*-----
```

*Continued on next page*

```
TRY.  
  SELECT * FROM (gv_tabname) INTO TABLE <fs_table>  
    UP TO pa_nol ROWS.  
  CATCH cx_sy_dynamic_osql_error.  
    MESSAGE e061.  
  ENDTRY.  
  
  CALL METHOD cl_bc402_dys_rtti_headers=>write_headers  
    EXPORTING  
      iv_tabname      = gv_tabname  
    EXCEPTIONS  
      type_not_found = 1  
      no_structure   = 2  
      no_ddic_type   = 3.  
  
  CASE sy-subrc.  
    WHEN 1.  
      MESSAGE e050 WITH gv_tabname.  
    WHEN 2.  
      MESSAGE e051 WITH gv_tabname.  
    WHEN 3.  
      MESSAGE e052 WITH gv_tabname.  
  ENDCASE.  
  
  cl_bc402_dys_gen_types=>write_any_table( <fs_table> ).
```



## Lesson Summary

You should now be able to:

- Query the properties of data objects and data types at runtime
- Query the properties of classes and instances at runtime

# Lesson: Generating Objects, Data Objects, and Data Types at Runtime

## Lesson Overview

You should be familiar with method for generating objects (that is, instances of classes) at runtime from object-oriented programming. In this lesson, you will learn a similar method for generating data objects (as instances of data types) at runtime. In contrast to data objects that are created statically with the DATA statement, the type of dynamically generated data objects does not have to be set directly, but instead can be determined dynamically at runtime.

Lastly, you will learn about runtime type creation (RTTC), a technique that lets you define the data type at runtime.



## Lesson Objectives

After completing this lesson, you will be able to:

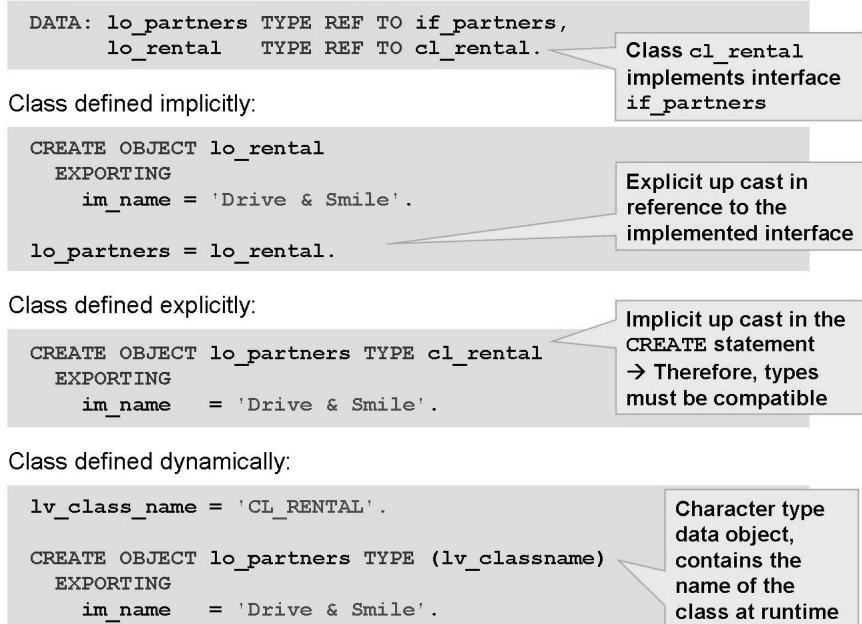
- Generate objects (instances) at runtime
- Generate data objects at runtime
- Generate data types at runtime

## Business Example

You have been asked to develop a flexible application. In addition to accessing existing data objects dynamically and analyzing the underlying data types, you also want to generate objects, data objects, and data types at runtime according to the program's needs. Accordingly, you want to learn about the available techniques.

## Generating Objects (Instances) at Runtime

The CREATE object generates objects (or more specifically, instances of classes) at runtime. Normally, the type of object - that is, the class to be instantiated - is defined implicitly through the static type of the reference variable used.



**Figure 168: Options for Generating Objects at Runtime**

Alternatively, you can use the TYPE addition to specify the class to be instantiated in the CREATE OBJECT statement. The following applies here:

Only a class can appear after the TYPE addition, not an interface

It must be possible to instantiate the class, which means it cannot be flagged as abstract

The static type of the reference variable must be compatible with the class

The last rule is true because an implicit up cast is carried out if necessary when the type is specified explicitly. The reference variable is compatible if one of the following statements applies:

The reference variable is typed with the class to be instantiated

The reference variable is typed with a superclass of the class to be instantiated

The reference variable is typed with an interface that is implemented by the class to be instantiated

The reference variable is typed with the generic REF TO OBJECT type.

The syntax for specifying an explicit type can include a data object in parentheses after TYPE. This enables dynamic selection of the class to be instantiated.



**Hint:** If classes are available for selection that are not linked through a shared superclass or the same implementing interface, you have to type the reference variable completely generically with TYPE REF TO OBJECT.

If all potential classes have the same constructor signature, the parameters can be supplied with static data (see above). However, it is much more likely that the parameters and exceptions of the constructors will differ. In this case, dynamic parameter transfer is needed:

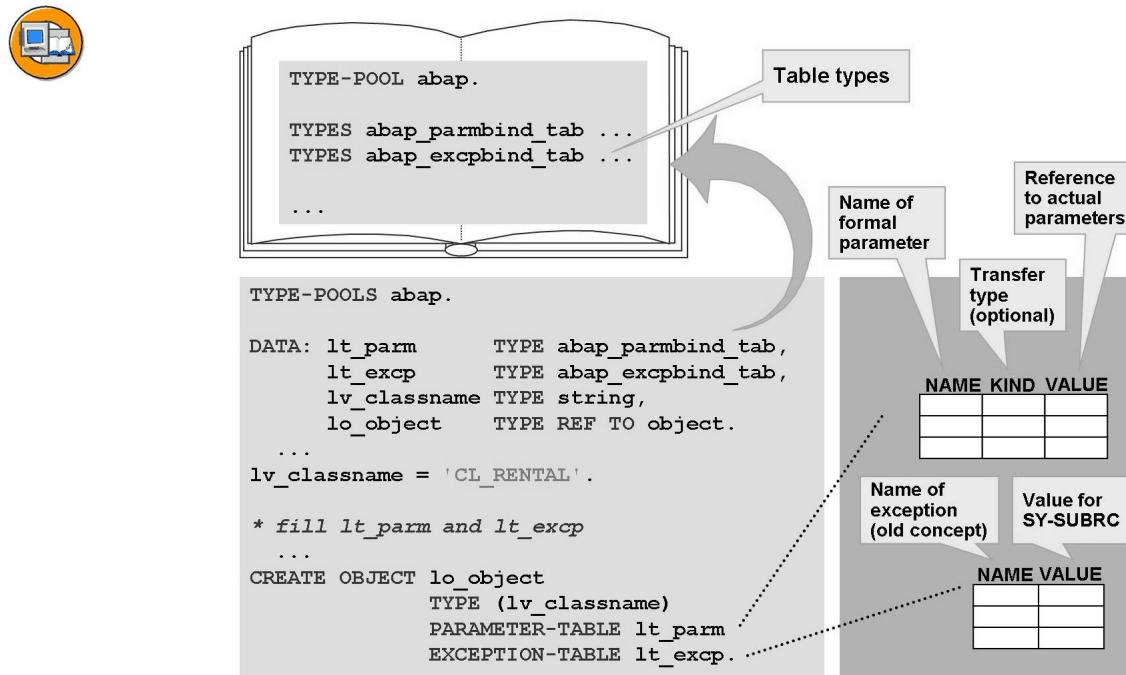


Figure 169: Dynamic Passing of Parameters to the Constructor

Like with dynamic method calls, this is done using internal tables after the PARAMETER-TABLE and EXCEPTION-TABLE additions. These tables must be typed and filled just like with method calls.

## Creating Data Objects at Runtime

You can also create data objects at runtime. The corresponding statement, CREATE DATA, has a similar definition to that of the CREATE OBJECT statement.

The CREATE DATA statement creates a data object dynamically, assigns memory to it, and points the general data reference variable to that data object in memory.



The data object is not given a name – which is why it is often called an **anonymous** data object. To address it, you have to deference the reference variable at runtime.



**Hint:** In contrast to the static DATA declaration, the memory for the data objects is not reserved until runtime, and not when the respective program is loaded. Like with class instances, the *garbage collector* releases the memory again as soon as the last reference to the data object is deleted.

```
DATA: lr_data  TYPE REF TO data,
      lr_spfli TYPE REF TO spfli.
```

Data type defined implicitly:

```
CREATE DATA lr_spfli.
lr_data = lr_spfli.
```

Explicit up cast in generic data reference

Data type defined explicitly:

```
CREATE DATA lr_data TYPE spfli.
CREATE DATA lr_data TYPE p LENGTH 3 DECIMALS 2.
CREATE DATA lr_data TYPE TABLE OF spfli.
CREATE DATA lr_data LIKE lv_dataobject.
```

Implicit up cast in the CREATE statement

Reference to any fully typed data object

Data type defined dynamically:

```
lv_type_name = 'SPFLI'.
CREATE DATA lr_data TYPE (lv_type_name).
CREATE DATA lr_data TYPE TABLE OF (lv_type_name).
```

Character type data object, contains the name of the data type at runtime

**Figure 170: Options for Creating Data Objects at Runtime**

Like with CREATE OBJECT, you can define the type of the data object implicitly, using the type of the reference variable, or explicitly, using the TYPE addition. If you use explicit typing, you can use a number of variants that you should be familiar with from DATA declarations: the use of built-in types, the implicit construction of table types, and reference to an existing data object with LIKE.

If you use character-type data objects in parentheses, you can define the data type for the generated data objects dynamically as well.



**Note:** You can also use the TYPE HANDLE addition to refer to a type description object from the RTTI.

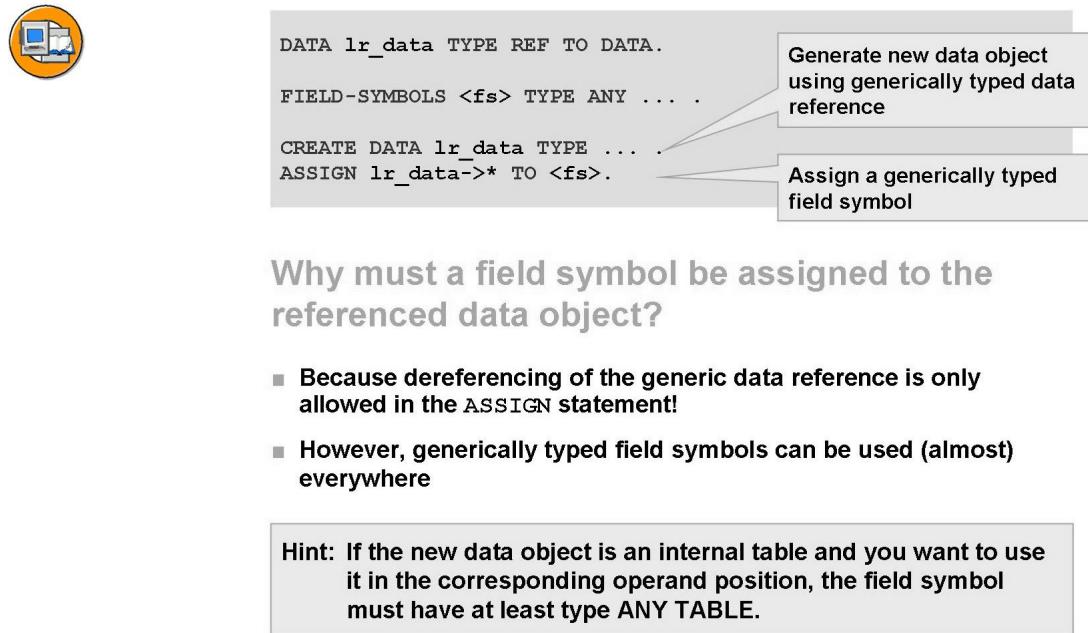


Figure 171: Access to Dynamically Generated Data Objects

If the data type is determined dynamically, the reference variable has to be typed generically: with TYPE REF TO DATA, currently the only generic type that supports data references.

As we learned in a previous lesson, generic data references can only be dereferenced in ASSIGN statements. As a result, CREATE DATA statements with generic reference variables are nearly inevitably followed by ASSIGN statements with field symbols typed more or less generically.

The following example shows a dynamic database access in which the suitable data object is generated at runtime.



```

DATA: lv_tablename      TYPE          string,
      lv_max_rows     TYPE          i,
      lr_table        TYPE REF TO data.

FIELD-SYMBOLS: <lt_table> TYPE ANY TABLE.

lv_tablename = 'SPFLI'.
lv_max_rows = 100.

CREATE DATA lr_table
      TYPE TABLE OF (lv_tablename).

ASSIGN lr_table->* TO <lt_table>.

SELECT * FROM (lv_tablename)
      UP TO lv_max_rows ROWS
      INTO TABLE <lt_table>.

```

Dynamic generation of an internal table with line type matching for DB table

Assignment of generically typed field symbol for usage in the SELECT statement

**Figure 172: Example: Dynamically Generated Internal Table and Dynamic SELECT**

## Runtime Type Creation (RTTC)

RTTI (runtime type identification) enables you to describe data types and object types with instances of RTTI classes, or type description objects. The RTTI concept was enhanced in *SAP Web AS 6.40*: In addition to describing existing types, you can now use it to generate new types. This enhanced function set is called RTTC (runtime type creation).

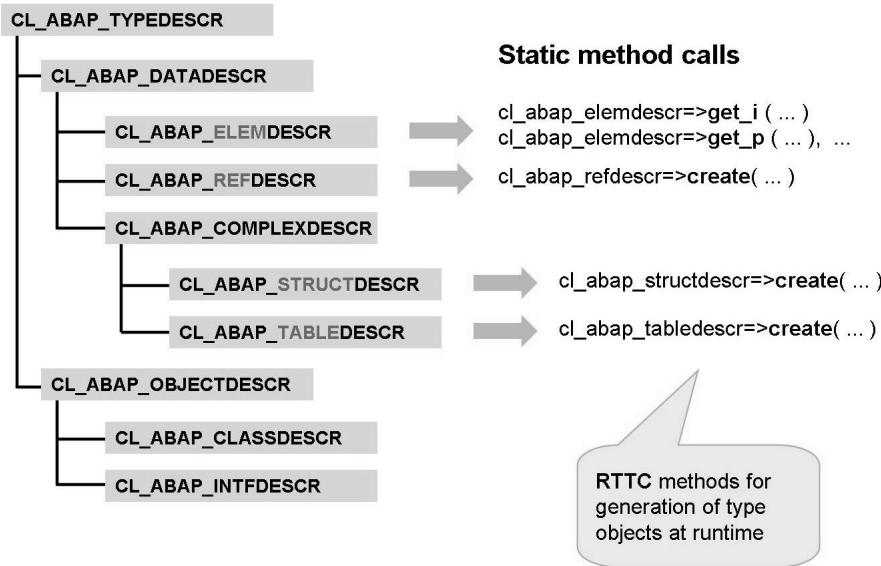
RTTC consists of a number of additional static methods in the RTTI description classes.



**Hint:** Note that type generation is restricted to data types. You cannot generate classes or interfaces dynamically.



### RTTC – Run Time Type Creation



**Figure 173: Dynamic Type Creation Methods of Runtime Type Services**

The `CL_ABAP_REFDESCR`, `CL_ABAP_STRUCTDESCR`, and `CL_ABAP_TABLEDESCR` classes were all enhanced with a `CREATE` method, while the `CL_ABAP_ELEMDESCR` class received an appropriate `GET` method for each built-in elementary ABAP type (`GET_I`, `GET_F`, `GET_C`, `GET_STRING`, and so on).

These methods return pointers to an instance of the respective class. In contrast to the `DESCRIBE` methods of the root class, the attribute values of this instance do not come from analyzing a data object, a reference, or a named type. Instead, they are passed on to the RTTC method explicitly.

The method signatures have been designed accordingly. While the `GET_I` method does not need any other input (type `i` is complete), method `GET_P` has two import parameters for length and the number of decimal places.

The `CREATE` method for reference types expects the pointers to the description objects for those reference types, while the `CREATE` method for structure types expects a list with names and types of the components. When table types are generated, a reference to an RTTI object that describes the line type is expected (among other things).

The following diagram shows how you can generate a table type dynamically with RTTC:

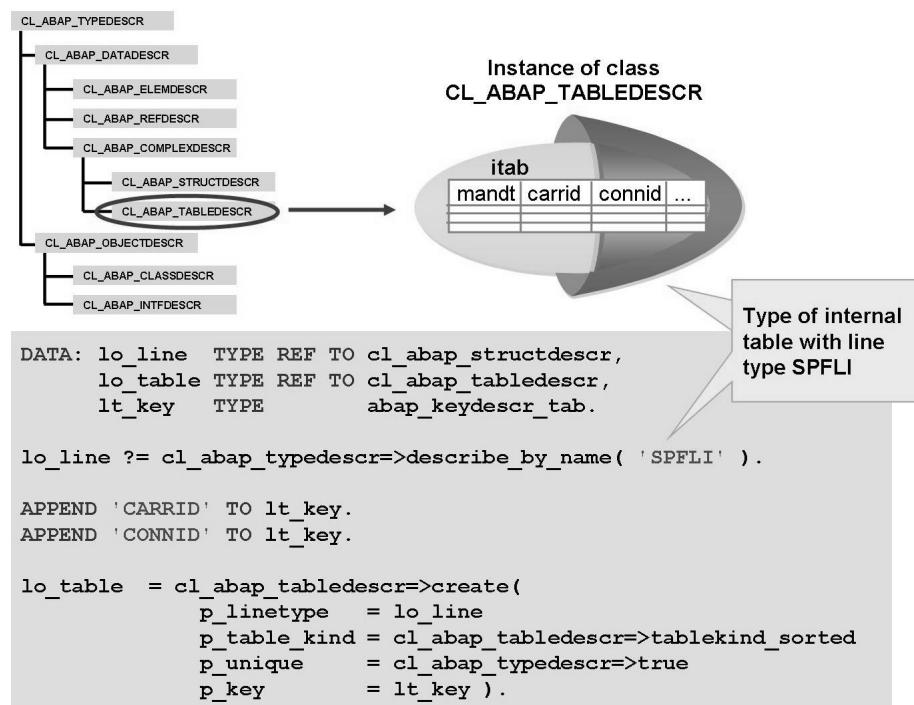


Figure 174: RTTC - Creating a Table Type

The only required parameter for the CREATE method is P\_LINE\_TYPE. It is typed with REF TO CL\_ABAP\_DATADESCR and has to be supplied with a reference to any description object for a data type. In the example provided here, SPFLI is used as the line type. The DESCRIBE\_BY\_NAME method returns the matching description object.

The optional parameters are used to define the table type and key. If you do not specify them, a standard table with a non-unique standard key is used.

The dynamically generated type can then be used in the CREATE DATA statement to create a data object with this specific type dynamically.



```

DATA: lv_tabname      TYPE      string,
      lv_max_rows   TYPE      i,
      lo_struct     TYPE REF TO cl_abap_structdescr,
      lo_table      TYPE REF TO cl_abap_tabledescr,
      lr_table      TYPE REF TO data.

FIELD-SYMBOLS: <lt_table> TYPE ANY TABLE.

lv_tabname = 'SPFLI'.
lv_max_rows = 100.

lo_struct ?= cl_abap_typedescr=>describe_by_name(
                           p_name = lv_tabname).
lo_table  = cl_abap_tabledescr=>create(
                           p_line_type = lo_struct).

CREATE DATA lr_table TYPE HANDLE lo_table.
ASSIGN lr_table->* TO <lt_table>.

SELECT * FROM (lv_tabname)
              UP TO lv_max_rows ROWS
              INTO TABLE <lt_table>.

```

Explicit construction  
of table type via RTTC

Usage of table type when  
generating the data object

**Figure 175: Generating an Internal Table with a Dynamically Created Type**

If you use the **HANDLE** addition with the **CREATE DATA** statement, this generates a data object whose type matches the specified type description object. The **HANDLE** must be specified as a reference variable with static type **CL\_ABAP\_DATADESCR** or one of its subclasses. The type object may have been created based on the existing data types, using RTTI methods, or by dynamically defining a new data type.



# Exercise 16: Generating Data Objects at Runtime

## Exercise Objectives

After completing this exercise, you will be able to:

- Generate and use data objects at runtime
- Use generically typed reference variables

## Business Example

You have been asked to develop a simple program that can display the contents of any database table. To do so, you want to find out about the options available in dynamic programming.

Your program can already output the content of any internal table in a list. The database access (the SELECT statement) already has a dynamic design. There is just one more thing separating you from your goal of reading any table: The data objects that you fill in the SELECT statement must have static definitions.

In this exercise, you will use the possibility of generating data objects dynamically at runtime. This will enable your program to read and display the content of any database table.

### Template:

BC402\_DYS\_RTTI\_HEADERS

### Solution:

BC402\_DYS\_CREATE\_DATA

### Task 1:

Copy program BC402\_DYS\_RTTI\_HEADERS (or your own program, ZBC402\_##\_RTTI\_HEADERS) to the name **ZBC402\_##\_CREATE\_DATA**, where ## is your group number. Familiarize yourself with the program and how it works.



**Hint:** If you did not do the exercise on RRTI, you can also simply copy your ZBC402\_##\_GEN\_DATA program. The display of the column headers is not relevant to this exercise.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Activate and test the program. Analyze the source code. Which data objects are filled during the database access? Which types do they have? Were they created statically or dynamically?

---

---

---

## Task 2:

Replace the two statically defined internal tables with a data object that you generate after the CASE structure (standard table with non-unique standard key). Define the type of the data object dynamically, so the line type precisely matches the structure of the database table you want to read.

1. Define a generically typed data reference (suggested name: `gr_data`).
2. Use the `gr_data` data reference to generate an internal table immediately after the ENDCASE statement. Access the contents of data object `gv_tabname` to define the line type of the table dynamically.
3. Move one of the ASSIGN statements outside of the WHEN blocks, after the generation of the data object, and delete the other one. Replace the statically defined internal table with the dynamically generated table.



**Hint:** To do so, you have to dereference the generic data reference.

4. Why can't you continue using `gr_data` directly?

---

---

---

5. Delete the definitions of the two static internal tables or flag them as comments.
6. Activate and test your program.

*Continued on next page*

### Task 3:

The static part of the program now only contains the filling of data object gv\_tabname with an appropriate literal for the name of the database table. Replace both checkboxes on the selection screen with a parameter for the name of any database table.

1. Delete the definitions parameters of the two checkboxes (or flag them as comments). Copy the user entry from the parameter directly to data object gv\_tabname instead of evaluating the checkboxes in a CASE structure.
2. Create a new parameter (suggested name: **pa\_tab**).



**Hint:** If you use the global type dd02l-tabname, you can provide a value (F4) help function for the users.

3. Delete the CASE structure (or flag it as comments). In the same position, program a value assignment from pa\_tab to gv\_tabname.
4. Activate and test your program.

## Solution 16: Generating Data Objects at Runtime

### Task 1:

Copy program BC402\_DYS\_RTTI\_HEADERS (or your own program, ZBC402\_##\_RTTI\_HEADERS) to the name **ZBC402##CREATE\_DATA**, where ## is your group number. Familiarize yourself with the program and how it works.



**Hint:** If you did not do the exercise on RRTI, you can also simply copy your ZBC402##GEN\_DATA program. The display of the column headers is not relevant to this exercise.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Activate and test the program. Analyze the source code. Which data objects are filled during the database access? Which types do they have? Were they created statically or dynamically?

**Answer:** One of two internal tables, gt\_conn or gt\_cust, is filled, depending on the user entry. While the data objects assigned to field symbol <fs\_table> (which are therefore addressed in the SELECT statement) are determined dynamically, but the data objects themselves have static definitions (DATA statement).

### Task 2:

Replace the two statically defined internal tables with a data object that you generate after the CASE structure (standard table with non-unique standard key). Define the type of the data object dynamically, so the line type precisely matches the structure of the database table you want to read.

1. Define a generically typed data reference (suggested name: **gr\_data**).
  - a) See the source code excerpt from the model solution.
2. Use the gr\_data data reference to generate an internal table immediately after the ENDCASE statement. Access the contents of data object gv\_tabname to define the line type of the table dynamically.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

3. Move one of the ASSIGN statements outside of the WHEN blocks, after the generation of the data object, and delete the other one. Replace the statically defined internal table with the dynamically generated table.



**Hint:** To do so, you have to dereference the generic data reference.

- a) See the source code excerpt from the model solution.
4. Why can't you continue using gr\_data directly?

**Answer:** Because gr\_data has a generic type and ABAP only allows dereferenced access to generically typed data references in ASSIGN statements.

5. Delete the definitions of the two static internal tables or flag them as comments.
  - a) See the source code excerpt from the model solution.
6. Activate and test your program.
  - a) Carry out this step in the usual manner.

### Task 3:

The static part of the program now only contains the filling of data object gv\_tabname with an appropriate literal for the name of the database table. Replace both checkboxes on the selection screen with a parameter for the name of any database table.

1. Delete the definitions parameters of the two checkboxes (or flag them as comments). Copy the user entry from the parameter directly to data object gv\_tabname instead of evaluating the checkboxes in a CASE structure.
  - a) See the source code excerpt from the model solution.
2. Create a new parameter (suggested name: **pa\_tab**).



**Hint:** If you use the global type dd02l-tabname, you can provide a value (F4) help function for the users.

- a) See the source code excerpt from the model solution.
3. Delete the CASE structure (or flag it as comments). In the same position, program a value assignment from pa\_tab to gv\_tabname.
  - a) See the source code excerpt from the model solution.
4. Activate and test your program.

*Continued on next page*

- a) Carry out this step in the usual manner.

## Result

Source code excerpt from the model solution:

```
*&-----*
*& Report BC402_DYS_CREATE_DATA
*&
*&-----*
*&
*&
*&-----*
```

REPORT bc402\_dys\_create\_data MESSAGE-ID bc402.

**\*DATA:**

\* gt\_cust TYPE ty\_customers,  
 \* gt\_conn TYPE ty\_connections.

**DATA:**

gr\_data TYPE REF TO data.

**DATA:**

gv\_tabname TYPE string.

**FIELD-SYMBOLS:**

<fs\_table> TYPE ANY TABLE.

SELECTION-SCREEN COMMENT 1(80) text-sel.

**\*PARAMETERS:**

\* pa\_xconn TYPE xfeld RADIobutton GROUP tab DEFAULT 'X',  
 \* pa\_xcust TYPE xfeld RADIobutton GROUP tab .

**PARAMETERS** pa\_tab TYPE dd021-tabname DEFAULT 'SPEFLI'.

**PARAMETERS:**

pa\_nol TYPE i DEFAULT '100'.

START-OF-SELECTION.

\* specific part
\*-----\*

\* CASE 'X'.
\* WHEN pa\_xconn.

*Continued on next page*

```

*
*      gv_tabname = 'SPFLI'.
*      ASSIGN gt_conn TO <fs_table>.
*
*      WHEN pa_xcust.
*
*      gv_tabname = 'SCUSTOM'.
*      ASSIGN gt_cust TO <fs_table>.
*
*      ENDCASE.

* dynamic part
*-----*

gv_tabname = pa_tab.

CREATE DATA gr_data TYPE TABLE OF (gv_tabname).
ASSIGN gr_data->* TO <fs_table>.

TRY.
  SELECT * FROM (gv_tabname) INTO TABLE <fs_table>
    UP TO pa_nol ROWS.
  CATCH cx_sy_dynamic_osql_error.
    MESSAGE e061.
ENDTRY.

CALL METHOD cl_bc402_dys_rtti_headers=>write_headers
  EXPORTING
    iv_tabname      = gv_tabname
  EXCEPTIONS
    type_not_found = 1
    no_structure   = 2
    no_ddic_type   = 3.

CASE sy-subrc.
  WHEN 1.
    MESSAGE e050 WITH gv_tabname.
  WHEN 2.
    MESSAGE e051 WITH gv_tabname.
  WHEN 3.
    MESSAGE e052 WITH gv_tabname.
ENDCASE.

cl_bc402_dys_gen_types=>write_any_table( <fs_table> ).
```



## Exercise 17: Optional: Generating Data Objects at Runtime with Runtime Type Creation (RTTC)

### Exercise Objectives

After completing this exercise, you will be able to:

- Use RTTC to generate structure types at runtime
- Use RTTC to generate table types at runtime
- Generate data objects based on dynamic data types

### Business Example

You have developed a simple program that can display the contents of any database table. The users are enthusiastic, but want even more. The program now needs to allow users to select the specific columns they want to read and display.

You find function module BC402\_SELECT\_COMPONENTS, which displays all the components of a global structure type and enables users to select components.

In the following, use the options of runtime type creation (RTTC) to dynamically generate a table type that only contains the user-selected components in its lines.

#### Template:

CL\_BC402\_DYT\_RTTC (global class)  
BC402\_DYS\_CREATE\_DATA (executable program)

#### Solution:

CL\_BC402\_DYS\_RTTC (global class)  
BC402\_DYS\_RTTC (executable program)

### Task 1:

Display function module BC402\_SELECT\_COMPONENTS. Analyze the interface and test the functional scope of the function module.

1. Display the function module. Analyze the interface. Which parameters are there and what are their types?

---

---

---

2. Which exceptions can the function module raise and what purpose do they serve?

---

---

---

3. Test the function module.

### Task 2:

Copy program BC402\_DYS\_CREATE\_DATA (or your own program, ZBC402\_##\_CREATE\_DATA) to the name **ZBC402\_##\_RTTC**, where ## is your group number. Familiarize yourself with the program and how it works.

Implement a call of the function module before the dynamic generation of the data object. Pass on the table name that the user entered on the selection screen.

1. Copy the program and all its subcomponents.
2. Call the function module before the CREATE DATA statement. Create a suitably typed data object to supply the export parameter with data. Pass the contents of gv\_tablename on to the import parameter. React to exceptions with suitable error messages from message class BC402.

### Task 3:

Copy the template CL\_BC402\_DYT\_RTTC to the name **ZCL\_BC402##\_RTTC**, where ## is your group number. Familiarize yourself with signature of method create\_table\_type.

1. Copy the global class.

*Continued on next page*

2. Analyze the `create_table_type` method signature. Which parameters are defined? How are they typed?

---



---



---

### Task 4:

Implement the `create_table_type` method. Use the RTTI techniques to analyze the data type whose name is contained in the import parameter. Use a suitable navigation method to calculate a list with the names and RTTI type description objects for all components in the structure.

1. Call a suitable method of class `CL_ABAP_TYPEDESCR` to generate a type description object for the data type. Create a reference variable with a suitable type for the return value (suggested name: `lo_struct`).



**Hint:** For simplicity's sake, you can assume that it is a structure type in the *ABAP Dictionary* and perform a direct down cast to the corresponding reference variable.

2. Call the `get_components` method for the RTTI instance. Create a data object with a suitable type for the return value (suggested name: `lt_comps`).

### Task 5:

Change the list of components so that it only contains the components listed in import parameter `it_comp_names`. Based on this list, create a new structure type and then a new table type.

1. Implement a loop over the `lt_comps` component list.
2. Check whether the component name in each line is contained in import parameter `it_comp_names`.



**Hint:** For example, you could use a statement like `FIND ... IN TABLE` or `READ TABLE ... TRANSPORTING NO FIELDS`.

3. Remove the current line from `lt_comps` if the corresponding table name is not contained in `it_comp_names`.



**Hint:** The abbreviated syntax for index access in loops can be used here.

*Continued on next page*

4. Generate an RTTI type description object for a new structure type. To do this, call the CREATE method of class CL\_ABAP\_STRUCTDESCR. Pass the reduced component list on to the method. Create a data object with a suitable type for the return parameter (suggested name: `lo_struct_new`).
5. Generate an RTTI type description object for a new table type. To do this, call the CREATE method of class CL\_ABAP\_TABLEDESCR. Pass on the reference to the RTTI object for the new structure type. Leave all the optional parameters set to their default values. Transfer the result of the method call directly to return parameter `ro_table_def` of your `create_table_type` method.

## Task 6:

Go back to your executable program, ZBC402\_##\_RTTC. Call up your `create_table_type` method, to generate a table type according to the user's wishes.

Now use a new data type when you generate the data object dynamically.

Change the SELECT statement such that it only reads the fields from the database that are contained in the line type of the internal table.



**Hint:** The `write_headers` method still outputs the headers for all the columns in the table. You can either do without the headers completely or use the `write_headers_by_data` method from class `CL_BC402_DYS_RTTI_HEADERS`. This method analyzes the actual line type of an internal table and outputs the column headers for it.

1. Call method `create_table_type` in the program, directly after the call of function module BC402\_SELECT\_COMPONENTS. Pass on the name of the database table and the list of components that the user selected.
2. Change the type in the CREATE DATA statement. Use the TYPE HANDLE addition to use the newly generated table type.
3. Change the SELECT statement. Use the list of component names as a dynamic field list.



**Hint:** Alternatively, you can also use the INTO CORRESPONDING FIELDS OF TABLE addition.

4. Activate and test your program.

## Solution 17: Optional: Generating Data Objects at Runtime with Runtime Type Creation (RTTC)

### Task 1:

Display function module BC402\_SELECT\_COMPONENTS. Analyze the interface and test the functional scope of the function module.

1. Display the function module. Analyze the interface. Which parameters are there and what are their types?

**Answer:** There is one import parameter, iv\_tabname, with type csequence and one export parameter, et\_comp\_names, with type string\_table (standard table with string line type).

2. Which exceptions can the function module raise and what purpose do they serve?

**Answer:**

**TYPE\_NOT\_FOUND**

The passed on type is unknown

**NO\_STRUCTURE**

The passed on type is not a structure

**NO\_DDIC\_TYPE**

The passed on type is not a DDIC type

3. Test the function module.

- a) Carry out this step in the usual manner.

### Task 2:

Copy program BC402\_DYS\_CREATE\_DATA (or your own program, ZBC402\_##\_CREATE\_DATA) to the name **ZBC402\_##\_RTTC**, where ## is your group number. Familiarize yourself with the program and how it works.

Implement a call of the function module before the dynamic generation of the data object. Pass on the table name that the user entered on the selection screen.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.

*Continued on next page*

2. Call the function module before the CREATE DATA statement. Create a suitably typed data object to supply the export parameter with data. Pass the contents of gv\_tablename on to the import parameter. React to exceptions with suitable error messages from message class BC402.
  - a) See the source code excerpt from the model solution.

### Task 3:

Copy the template CL\_BC402\_DYT\_RTTC to the name **ZCL\_BC402\_##\_RTTC**, where ## is your group number. Familiarize yourself with signature of method create\_table\_type.

1. Copy the global class.
  - a) Carry out this step in the usual manner.
2. Analyze the create\_table\_type method signature. Which parameters are defined? How are they typed?

**Answer:** Import parameter iv\_tablename with generic type csequence, import parameter it\_comp\_names with type string\_table, and return parameter ro\_tabledescr, a reference to class cl\_abap\_tabledescr.

### Task 4:

Implement the create\_table\_type method. Use the RTTI techniques to analyze the data type whose name is contained in the import parameter. Use a suitable navigation method to calculate a list with the names and RTTI type description objects for all components in the structure.

1. Call a suitable method of class CL\_ABAP\_TYPEDESCR to generate a type description object for the data type. Create a reference variable with a suitable type for the return value (suggested name: **lo\_struct**).



**Hint:** For simplicity's sake, you can assume that it is a structure type in the *ABAP Dictionary* and perform a direct down cast to the corresponding reference variable.

- a) See the source code excerpt from the model solution.
2. Call the get\_components method for the RTTI instance. Create a data object with a suitable type for the return value (suggested name: **1t\_comps**).
  - a) See the source code excerpt from the model solution.

*Continued on next page*

## Task 5:

Change the list of components so that it only contains the components listed in import parameter `it_comp_names`. Based on this list, create a new structure type and then a new table type.

1. Implement a loop over the `lt_comps` component list.
  - a) See the source code excerpt from the model solution.
2. Check whether the component name in each line is contained in import parameter `it_comp_names`.



**Hint:** For example, you could use a statement like `FIND ... IN TABLE` or `READ TABLE ... TRANSPORTING NO FIELDS`.

- a) See the source code excerpt from the model solution.
  3. Remove the current line from `lt_comps` if the corresponding table name is not contained in `it_comp_names`.
- 
- 
- Hint:** The abbreviated syntax for index access in loops can be used here.
- a) See the source code excerpt from the model solution.
  4. Generate an RTTI type description object for a new structure type. To do this, call the `CREATE` method of class `CL_ABAP_STRUCTDESCR`. Pass the reduced component list on to the method. Create a data object with a suitable type for the return parameter (suggested name: `lo_struct_new`).
    - a) See the source code excerpt from the model solution.
  5. Generate an RTTI type description object for a new table type. To do this, call the `CREATE` method of class `CL_ABAP_TABLEDESCR`. Pass on the reference to the RTTI object for the new structure type. Leave all the optional parameters set to their default values. Transfer the result of the method call directly to return parameter `ro_table_def` of your `create_table_type` method.
    - a) See the source code excerpt from the model solution.

*Continued on next page*

## Task 6:

Go back to your executable program, ZBC402\_##\_RTTC. Call up your create\_table\_type method, to generate a table type according to the user's wishes.

Now use a new data type when you generate the data object dynamically.

Change the SELECT statement such that it only reads the fields from the database that are contained in the line type of the internal table.



**Hint:** The write\_headers method still outputs the headers for all the columns in the table. You can either do without the headers completely or use the write\_headers\_by\_data method from class CL\_BC402\_DYS\_RTTI\_HEADERS. This method analyzes the actual line type of an internal table and outputs the column headers for it.

1. Call method create\_table\_type in the program, directly after the call of function module BC402\_SELECT\_COMPONENTS. Pass on the name of the database table and the list of components that the user selected.
  - a) See the source code excerpt from the model solution.
2. Change the type in the CREATE DATA statement. Use the TYPE HANDLE addition to use the newly generated table type.
  - a) See the source code excerpt from the model solution.
3. Change the SELECT statement. Use the list of component names as a dynamic field list.



**Hint:** Alternatively, you can also use the INTO CORRESPONDING FIELDS OF TABLE addition.

- a)
4. Activate and test your program.
  - a) Carry out this step in the usual manner.

## Result

Source code excerpt from the model solution:

```
Method create_table_type (Class  
CL_BC402_DYS_RTTC)
```

```
METHOD create_table_type.
```

*Continued on next page*

```

DATA:
  lo_struct      TYPE REF TO cl_abap_structdescr,
  lo_struct_new  TYPE REF TO cl_abap_structdescr.

DATA:
  lt_comps TYPE cl_abap_structdescr=>component_table.

FIELD-SYMBOLS:
  <fs_comp> LIKE LINE OF lt_comps.

* get description of transparent table (=structure type)
lo_struct ?= cl_abap_typedescr=>describe_by_name( iv_tabname ).

* get list of components (including component types)
lt_comps = lo_struct->get_components( ).

LOOP AT lt_comps ASSIGNING <fs_comp>.
  FIND <fs_comp>-name IN TABLE it_comp_names.
  IF sy-subrc <> 0.
    DELETE lt_comps.
  ENDIF.

ENDLOOP.

* alternative solution with read table
* remove all components but the requested ones
* LOOP AT lt_comps ASSIGNING <fs_comp>.
*   READ TABLE it_comp_names
*     TRANSPORTING NO FIELDS
*       WITH TABLE KEY table_line = <fs_comp>-name.
*   IF sy-subrc <> 0.
*     DELETE lt_comps .
*   ENDIF.
*
* ENDLOOP.

* create new structure type with the remaining components
lo_struct_new = cl_abap_structdescr=>create(
  p_components = lt_comps ).

* create table type with this new structure type as line type
ro_tabledescr = cl_abap_tabledescr=>create(
  p_line_type  = lo_struct_new ).
```

*Continued on next page*

```
ENDMETHOD.
```

## Executable Program BC402\_DYS\_RTTC

```
REPORT bc402_dys_rttc MESSAGE-ID bc402.

DATA:
  gr_table  TYPE REF TO data.

DATA:
  gv_tabname  TYPE string.

DATA:
  gt_comp_names  TYPE string_table,
  go_table      TYPE REF TO cl_abap_tabledescr.

FIELD-SYMBOLS:
  <fs_table>  TYPE ANY TABLE.

SELECTION-SCREEN COMMENT 1(80) text-sel.

PARAMETERS pa_tab TYPE dd02l-tablename DEFAULT 'SPFLI'.

PARAMETERS:
  pa_nol  TYPE i DEFAULT '100'.

START-OF-SELECTION.

  gv_tabname = pa_tab.

  CALL FUNCTION 'BC402_SELECT_COMPONENTS'
    EXPORTING
      iv_tabname      = gv_tabname
    IMPORTING
      et_comp_names  = gt_comp_names
    EXCEPTIONS
      type_not_found = 1
      no_structure   = 2
      no_ddic_type   = 3.

  CASE sy-subrc.
```

*Continued on next page*

```

WHEN 1.
  MESSAGE e050 WITH gv_tabname.
WHEN 2.
  MESSAGE e051 WITH gv_tabname.
WHEN 3.
  MESSAGE e052 WITH gv_tabname.
ENDCASE.

go_table = cl_bc402_dys_rttc->create_table_type(
  iv_tabname      = gv_tabname
  it_comp_names = gt_comp_names ).

CREATE DATA gr_table TYPE HANDLE go_table.
ASSIGN gr_table->* TO <fs_table>.

TRY.
  SELECT (gt_comp_names) FROM (gv_tabname)
    INTO TABLE <fs_table>
    UP TO pa_nol ROWS.
  CATCH cx_sy_dynamic_osql_error.
    MESSAGE e061.
ENDTRY.

CALL METHOD cl_bc402_dys_rtti_headers->write_headers_by_data
  EXPORTING
    it_table          = <fs_table>
  EXCEPTIONS
    no_structure      = 1
    no_ddic_type      = 2
    component_not_elem = 3.

CASE sy-subrc.
  WHEN 1.
    MESSAGE e050 WITH gv_tabname.
  WHEN 2.
    MESSAGE e051 WITH gv_tabname.
  WHEN 3.
    MESSAGE e053 WITH gv_tabname.
ENDCASE.

cl_bc402_dys_gen_types->write_any_table( <fs_table> ).
```



## Lesson Summary

You should now be able to:

- Generate objects (instances) at runtime
- Generate data objects at runtime
- Generate data types at runtime



## Unit Summary

You should now be able to:

- Outline the various techniques for dynamic programming available in ABAP
- Dynamically define ABAP syntax components (tokens)
- Define parts of SELECT statements at runtime
- Call function modules dynamically
- Call transactions and programs dynamically
- Generate a complete program dynamically
- Explain what generic types are and what they are used for
- Use generically typed parameters, field symbols, and data references
- Use field symbols to access data objects dynamically
- Use field symbols to access attributes and structure components dynamically
- Query the properties of data objects and data types at runtime
- Query the properties of classes and instances at runtime
- Generate objects (instances) at runtime
- Generate data objects at runtime
- Generate data types at runtime



I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

Internal Use SAP Partner Only

Internal Use SAP Partner Only

# Unit 5

## ABAP Open SQL in Detail

### Unit Overview

*Open SQL* features a variety of options that go beyond the simple SELECT statements you learned about in course BC400. We examine these features in detail in this unit. They include sorting, formatting, and aggregating data directly in the database, as well as combining data from multiple database tables.



### Unit Objectives

After completing this unit, you will be able to:

- Name the relevant system components related to SAP Open SQL
- Describe the process flows involved in a database access using Open SQL
- Explain the following terms: database interface, SAP table buffer, database buffer, cursor cache, database index, and optimizer
- Explain the importance of the SAP database interface
- Explain the importance of the possible operators in WHERE conditions
- Use the different operators correctly
- Know the options available for specifying data objects after INTO
- Implement sequential processing of large data volumes
- Request sorted or aggregated data from the database
- Use aggregate functions correctly
- Use views and formulate joins correctly
- Understand and use subqueries
- Use the FOR ALL ENTRIES addition correctly
- Access sub sequences or properties of LOBs on the database table using locators.
- Use locators to copy LOBs on the database table without first copying the data to the application server.
- Process LOBs sequentially by using the stream concept.

### Unit Contents

Lesson: Database, Database Interface, and Open SQL .....	363
--	-----

Lesson: WHERE Condition and Target Area.....	380
Lesson: SELECT Statements: Processing and Aggregating Value Sets .....	390
Exercise 18: SELECT Statements – Aggregating Value Sets .....	399
Lesson: Reading from Multiple Database Tables.....	409
Exercise 19: Implementing a Join for Three Tables.....	421
Exercise 20: Optional: Buffering Data Completely or On Demand ..	429
Exercise 21: Optional: Read Additional Data with the FOR ALL ENTRIES Addition.....	439
Lesson: Streams and Locators .....	448

# Lesson: Database, Database Interface, and Open SQL

## Lesson Overview

This lesson gives you an overview of the system architecture and process flows involved in database accesses with Open SQL.

You learn about secondary indexes and the optimizer function in the database, and why they are important to efficient data access.

You find out what the database interface does and how SAP table buffering is used to reduce the load on the database and speed up access to frequently needed data.



## Lesson Objectives

After completing this lesson, you will be able to:

- Name the relevant system components related to SAP Open SQL
- Describe the process flows involved in a database access using Open SQL
- Explain the following terms: database interface, SAP table buffer, database buffer, cursor cache, database index, and optimizer
- Explain the importance of the SAP database interface

## Business Example

You want to implement database accesses with Open SQL in your applications. These database accesses must be sophisticated and efficient. In a first step, find out about the details of the system architecture and the process flows involved in database accesses.

## Architecture

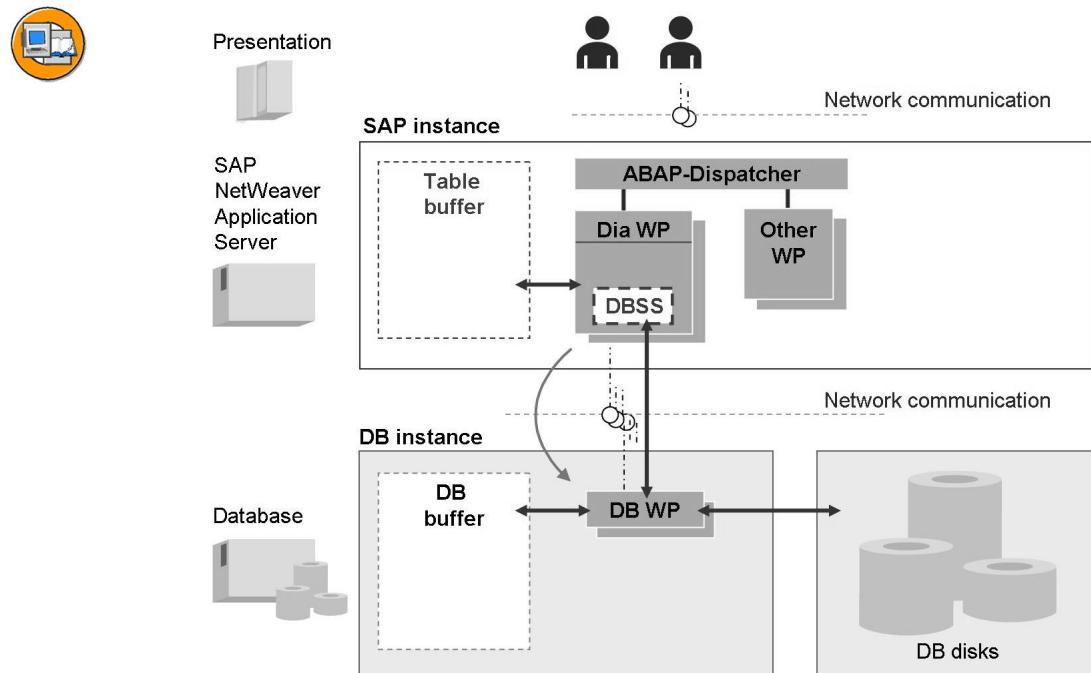


Figure 176: Architecture of the SAP NetWeaver Application Server (ABAP)

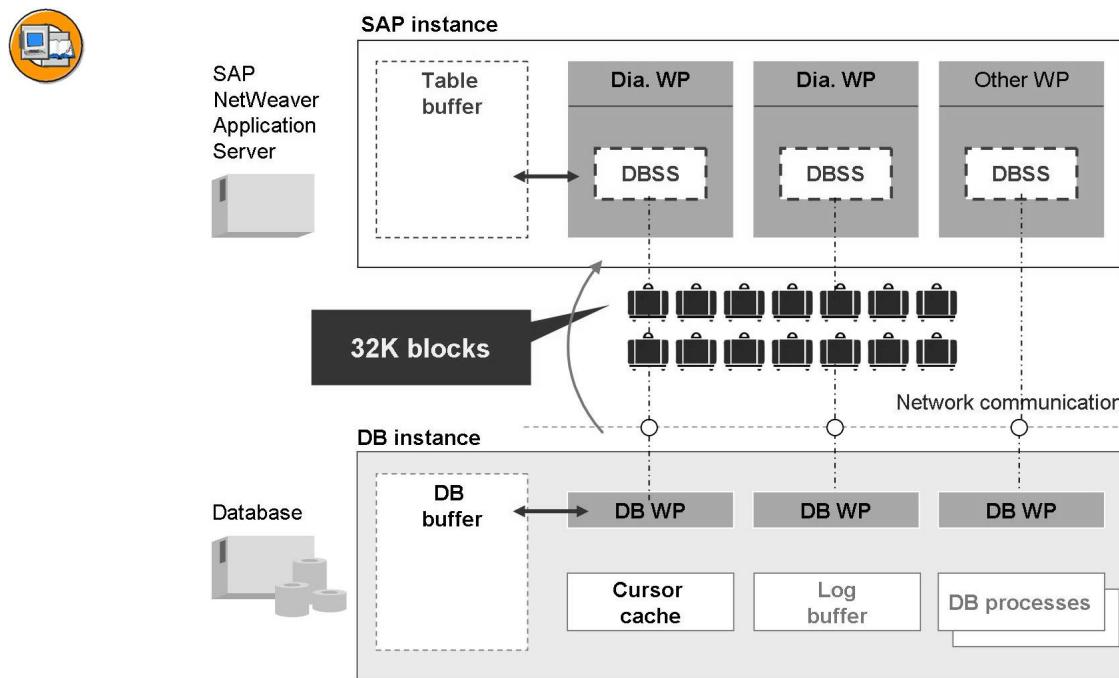
The SAP system is implemented as a multitier client/server architecture (presentation layer, application layer, and database layer).

Data transport between the application layer and the database is the responsibility of the **SAP database interface (DBSS)**. It accepts the SQL statements that the ABAP applications send in the dialog work processes (Dia-WP), retrieves the requested data, and makes it available to the application program. The application program can contain statements in **ABAP Open SQL** and **native SQL** (SQL of the underlying database). Accordingly, the DBSS is divided into an ABAP Open SQL interface and a native SQL interface.

Tasks of the ABAP Open SQL interface:

- Translating ABAP Open SQL to native SQL
- Client handling (including the client in the WHERE condition)
- Managing the SAP table buffer (frequently needed data is buffered automatically in the working memory of the application server and not read over the network from the database each time).

The database contains several database work processes (DB-Wp, or “shadow processes”), which process the requests from the SAP application servers. Each work process on the application server side is assigned to exactly one work process in the database.



**Figure 177: Database Architecture and Data Transport**

The data is stored persistently in the file system of the database server and is loaded from the file system to the DB work processes when requests from the database service processes are received. Since the operation can take long time, a database cache (DB buffer in the sketch) is available for storing frequently used data. The data in this buffer can be accessed much faster than it can be retrieved from the file system of the database server.

Aside from the database buffer, the database allocates another shared memory area, the **DB cursor cache** (database access paths), which contains redo log (and other) information.



**Hint:** Note that ABAP developers have no influence on the buffering mechanisms within the DB server. In contrast, ABAP developers use settings in the *ABAP Dictionary* to define which tables can be stored in the *SAP table buffer*.

In addition to the database work processes, there are also other database services for various purposes (starting communications, changing database tables, locking mechanism, archiving, and so on).

## Database Indexes and Database Optimizer

When processing a specific database access, it is critically important that the database determines the addressed records and provide them to the DB-Wp as quickly as possible. To this end, the database manages one or more **indexes** for each database table, in addition to its actual contents. An index consists of selected fields of the database table that are stored in the database copy as a sorted table; these fields refer to the actual table lines. When a table is accessed the database determines whether it accesses the contents of the database table directly, or reads an index first and then determines the actual data through the references in the index. A combination of multiple indexes is also possible. A variety of factors influences the speed of data delivery for specific SQL statements. The database feature that weighs all these factors and determines the access strategy (and thus whether indexes are used) is called an **optimizer**.

### The Database Optimizer

A database optimizer is a database feature that analyzes an SQL statement and then determines an access strategy dependent on the WHERE condition and the structure of the existing indexes. It is the database optimizer that determines whether one of the existing indexes is used, and if so, which one.

Even if all the fields in an index are valued, it is not certain that the optimizer selects that index. There is always a chance, for example, that the optimizer selects a sequential search of all the table contents instead.



### The Database Optimizer

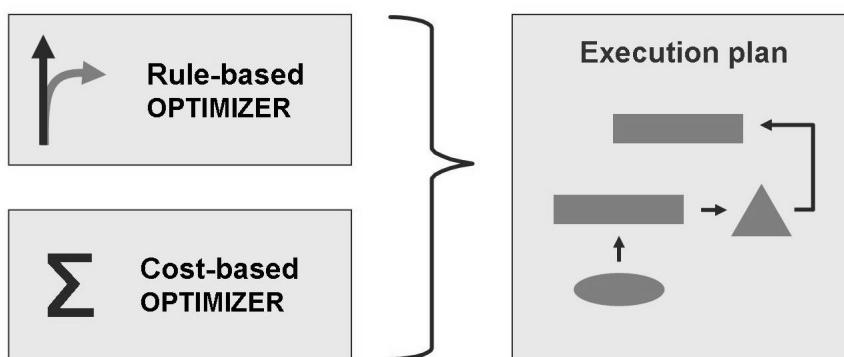


Figure 178: The Database Optimizer

You can distinguish between two characteristic values in the way the Optimizer works:

A **rule-based** optimizer analyzes the structure of an SQL statement (mainly the SELECT and WHERE conditions without values), and the index structure of the table (or tables). It then uses a set of analysis rules to decide the most appropriate procedure for executing the statement. The specific contents of the table are irrelevant.

A **cost-based** optimizer also analyzes values in the WHERE condition and the table statistics. The statistics contain low and high values for the fields, or, in some cases, a histogram of the distribution of the data in the table. Cost-based optimizers use more information about the table, and usually produce faster access. One of the disadvantages is that the statistics must be periodically updated.

### Setting Up Indexes

When you create a database table in the *ABAP Dictionary*, you have to specify the combination of fields that uniquely identifies an entry within the table. You place these fields at the start of the table. They are called **key fields (primary keys)**. On most database systems, the key fields are used to create an index in addition to the table (all others). This index is called the **primary index**. It is **unique** by definition.

 **Note:** In contrast to other database systems, the primary index and the actual table data are a single unit on Microsoft SQL Server. Therefore, it is not possible to access the data without using the primary index in this database. The following discussions are based on database systems where the primary index is separated from the table data.

In addition to the primary index, you can define one or more **secondary indexes** for a table in the *ABAP Dictionary* and save them in the database. Secondary indexes can be either unique or non-unique.

Both index records and table records are organized in data blocks. When an ABAP program sends an SQL statement to the database, the requested data records are searched either in the database table itself (full table scan) or with the assistance of an index (index unique scan or index range scan). If all the requested fields are contained in the index, the table itself does not have to be accessed.

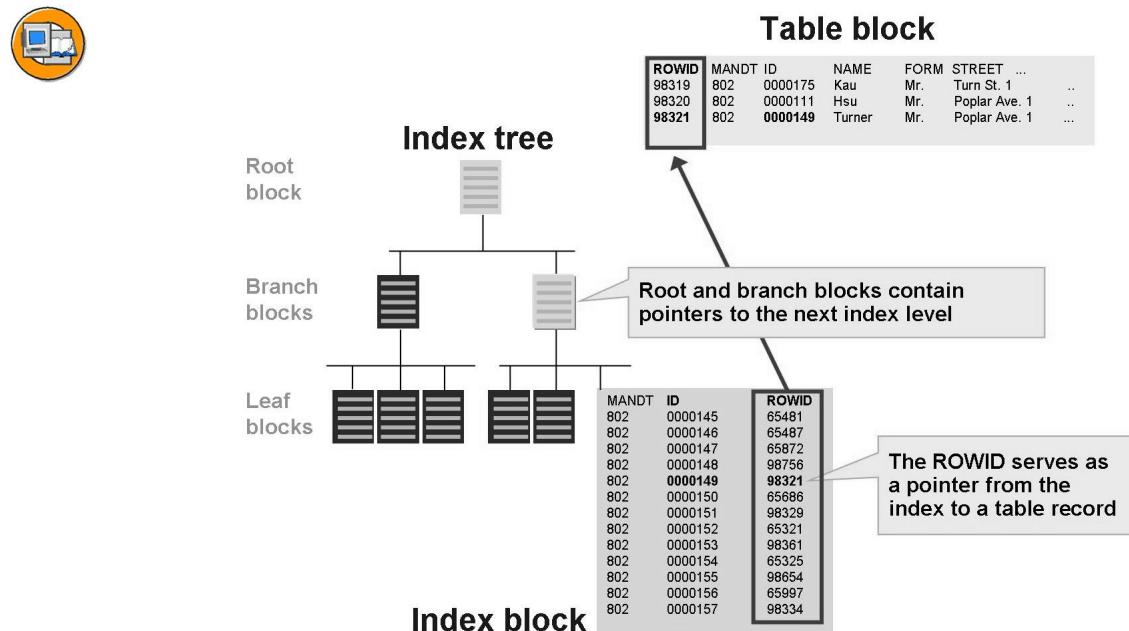


Figure 179: The Index Tree

Data blocks can be index blocks or table blocks. They represent the granularity with which data can be written to or read from the hard drive. Data blocks can contain several data records; conversely, one record can also stretch over several data blocks.

The **root level** contains a single index block that contains pointers to the index blocks at **branch level**. The branch blocks either contain part of the index fields and pointers to the index blocks at **leaf level**, or all the index fields and a pointer to the table records organized in the table blocks. The index blocks at leaf level contain all index fields and pointers to the table records from the table blocks.

→ **Note:** The expression **ROWID** for the pointer, is specific for Oracle databases.

The index records are saved in the index tree sorted by index field. This is what makes index-based access faster in the first place. The table records in the table blocks are not sorted.

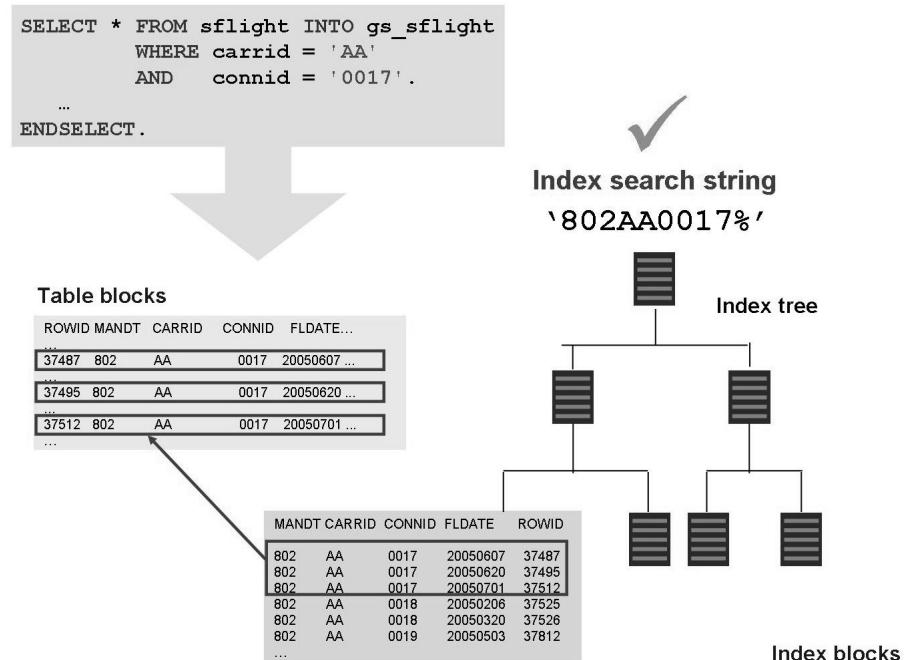


**Hint:** To prevent an index from requiring too much storage space, an index should not consist of too many fields. Using only a few, carefully selected fields promotes reusability and reduce the possibility that the database optimizer selects a processing-intensive access path.



## Using Indexes

When looking for a rough estimation of how useful an index is for a specific SELECT statement, you can have a look at the **index search string** for this statement: The index search string is formed from the WHERE condition. The desired values of the fields contained in the index are concatenated. The longer the search string is from the left without placeholders ("\_" or "%") the more efficient is the access through this index.

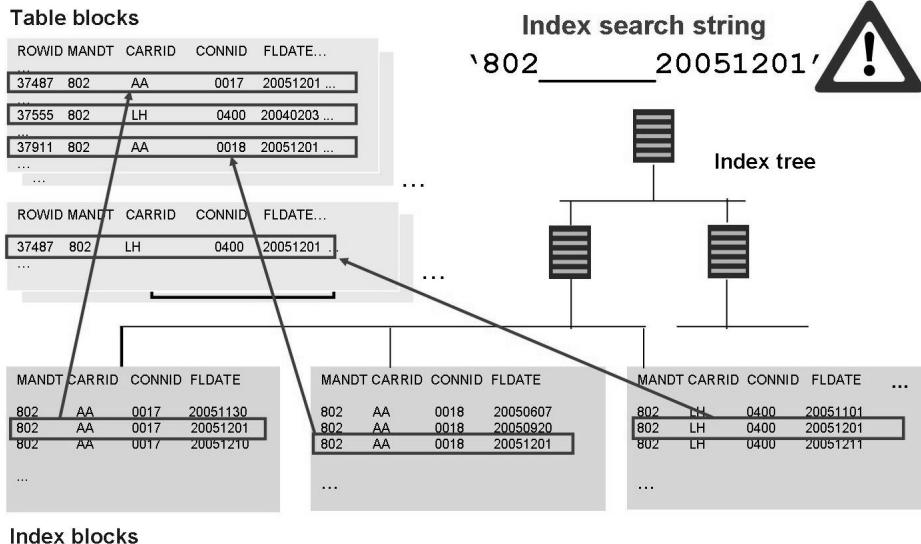


**Figure 180: The Index Search String (1)**

In this example, all three fields of the primary index (MANDT, CARRID, and CONNID) of table SFLIGHT are evaluated in the WHERE condition (the client is set automatically by the DBSS). The last field of the index, FLDATE; is not qualified. Therefore, the search string for the database is '802AA0017%'. Starting from the left, there are no gaps in the search string, until the last field. As a result, the database has to expend relatively little effort to search the index tree and can quickly return the result set.



```
SELECT * FROM sflight WHERE fldate = '20051201'. ... ENDSELECT.
```



**Figure 181: The Index Search String (2)**

In this example, the CARRID and CONNID fields are missing in the WHERE condition. The corresponding positions for the two fields are set to "\_" in the index search string. Therefore, the fully specified area that is used to select the index blocks only consists of the MANDT field. Since many index records fulfill the condition `MANDT = "802"`, many index blocks are read and their index records are checked. From the set of index records, all those that fulfill the condition `FDATE="20051201"` are filtered out.



**Hint:** You should qualify (fill with values) as many index fields as possible, from left to right, and not leave any gaps in the index search string. Ultimately, however, the application logic decides how the SELECT statement is coded. Accordingly, the above example may be entirely justified.

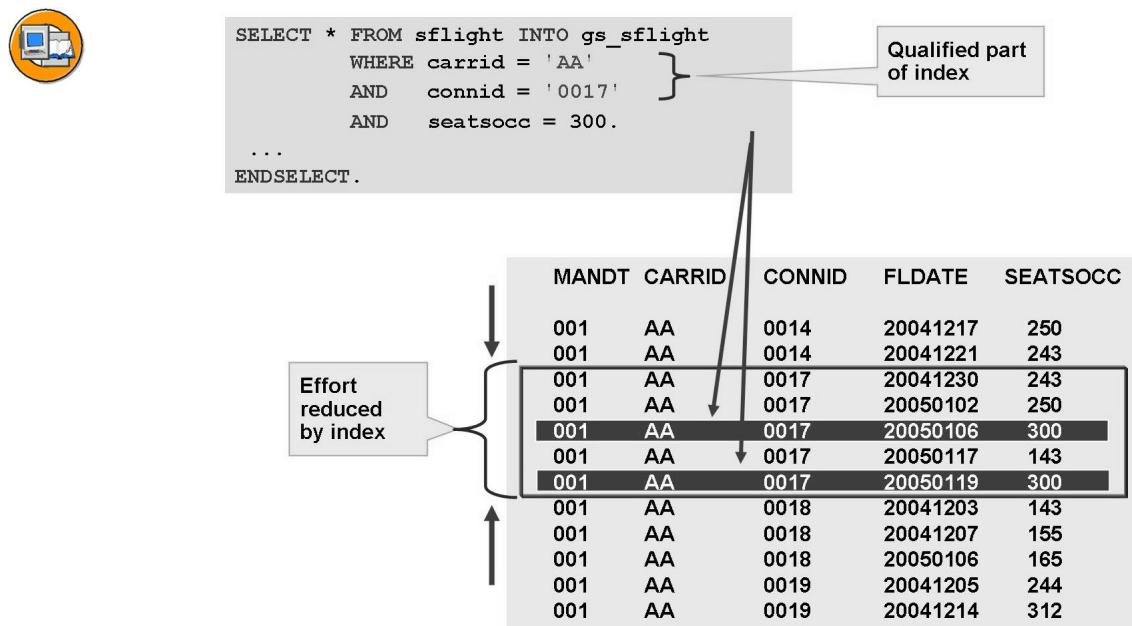


Figure 182: Reducing the Search Effort

The more index fields are qualified (or assigned values) from left to right, the less search effort the database has to make to determine the hit list. If you use the CARRID, CONNID, and SEATSOCC fields in the WHERE condition to access table SFLIGHT, this most likely results in a primary index access in which the first three index fields (including the client) are used. Since the SEATSOCC field is also specified, these exact records are read from the hit list and transferred to the application server. The search delivers excellent performance in this example, transferring exactly the records that the application really needs.

## SAP Table Buffering

The SAP table buffer is a memory area in shared memory of the active application server. You can store table contents in it as often as needed to avoid database accesses and conserve database resources (CPU load, main memory).

## The SAP Table Buffer

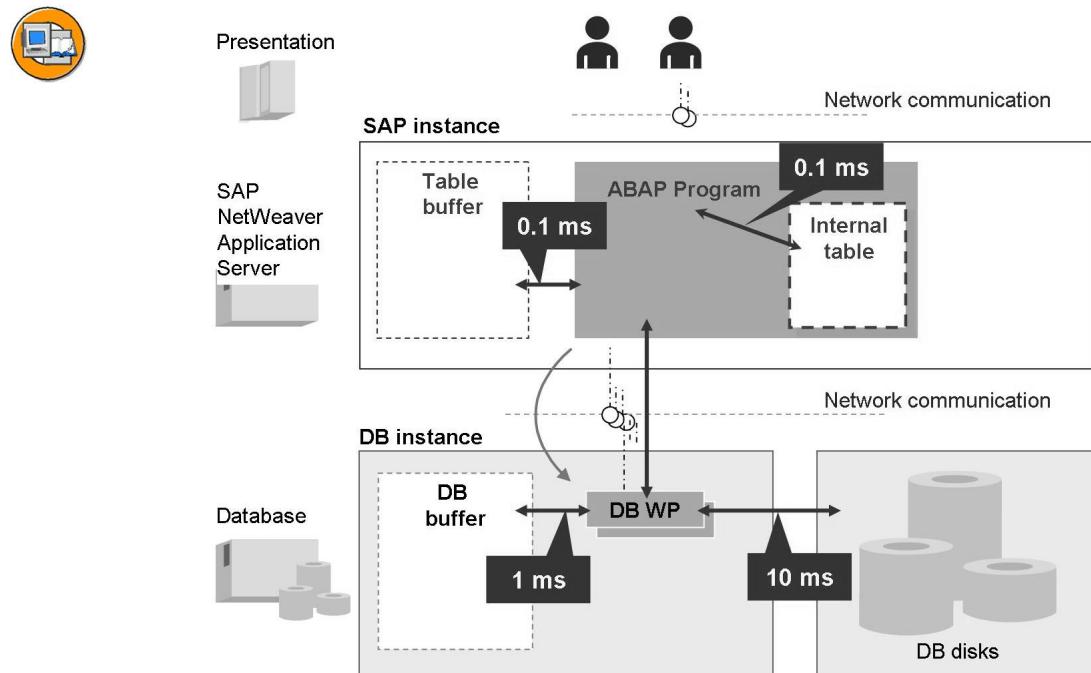


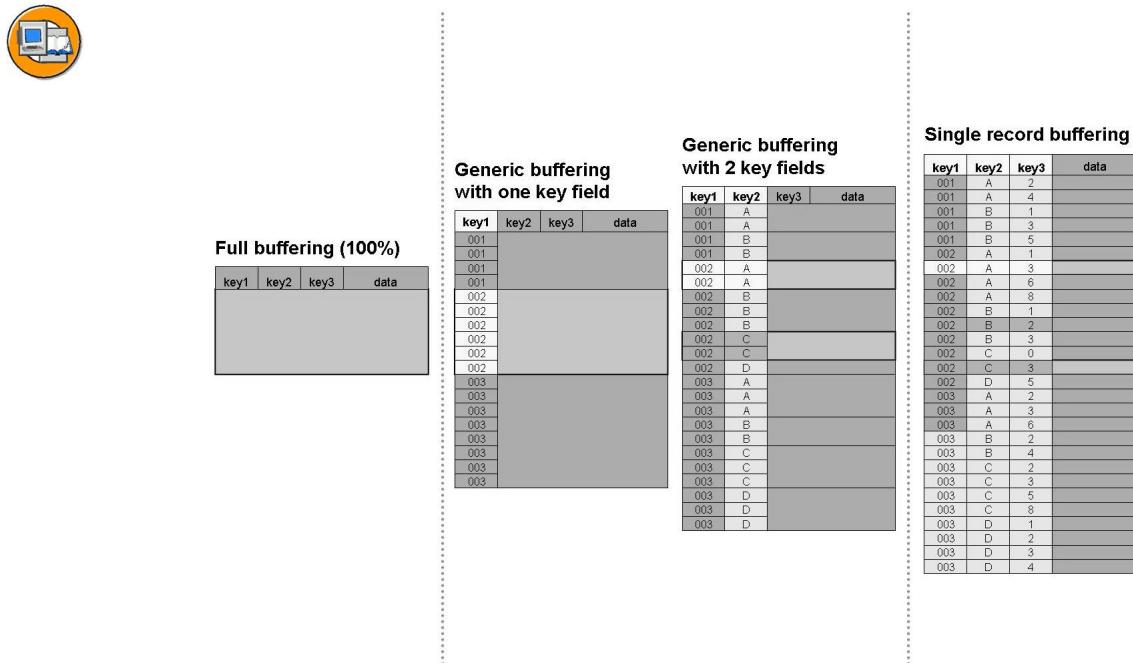
Figure 183: Table Buffering: Overview

Reducing the database load is important, because the database server is a central resource. In addition, accesses to the SAP table buffer are much faster than database accesses. If data is available in the database buffers on the SAP application servers, retrieval takes around 0.1 ms/record. When the records are read from the database data buffer, approx. 1 ms/record is needed. This means records buffered on the application server can be accessed up to 100 times faster than data in the database. As a result, the work processes on the application server encounter shorter wait times.

Whether and how a database table is buffered is determined during its definition. The database interface controls the filling and reading of the buffer at runtime and is usually effective during access using Open SQL statements, but can be bypassed by certain variants (see performance aspects of Open SQL).

### Buffering Types

When you define a database table in the *ABAP Dictionary*, you can allow, forbid, or require table buffering in the technical settings. If buffering is allowed, one of the following buffering types must be selected:



**Figure 184: Buffering Types**

We differentiate between the single record buffer (technically: TABLP), the generic table buffer (technically: TABL) and the buffer for complete tables (full table buffering). The single record buffer contains the records from the single record-buffered tables. Physically, the generic table buffer contains the records of the generically and fully buffered tables (technically, both of these buffering types use the same memory area on the application server).

### Full buffering

The first time the table is accessed, the entire table is loaded into the table buffer (generic table buffer). All subsequent accesses of the table can be served from the table buffer.

### Generic buffering

If you want to buffer a table generically, you first have to define the generic area. The generic area consists of the first n key fields of the table. If an SQL statement is executed with a specific instantiation of the generic area (such as `SELECT * FROM TAB1 WHERE KEY1 = '002'` or `SELECT * FROM TAB2 WHERE KEY1 = '002' AND KEY2 = 'A'`) for a generically buffered table, the corresponding records are loaded into the table buffer. All subsequent accesses with the same instantiation (and potentially additional fields not defining the generic area) can be served from the table buffer.

### Single record buffering

Only single records are read from the database and loaded into the table buffer (single record buffer).

 **Note:** The single record buffer is only read, if the SELECT SINGLE statement is used. The where clause has to contain restrictions for all key fields. Restrictions for non-key fields may be added.

Each buffering type can be considered to be a variant of generic buffering with n key fields:

- Full buffering: n = 0
- Generic buffering:  $0 < n <$  number of key fields
- Single record buffering: n = number of key fields

### Buffer Synchronization

Buffer synchronization is a mechanism that the application servers in a system use to ensure that obsolete data is removed from the table buffers when data in a buffered table is changed. This synchronization is performed in fixed intervals, which you can configure in the system profile.

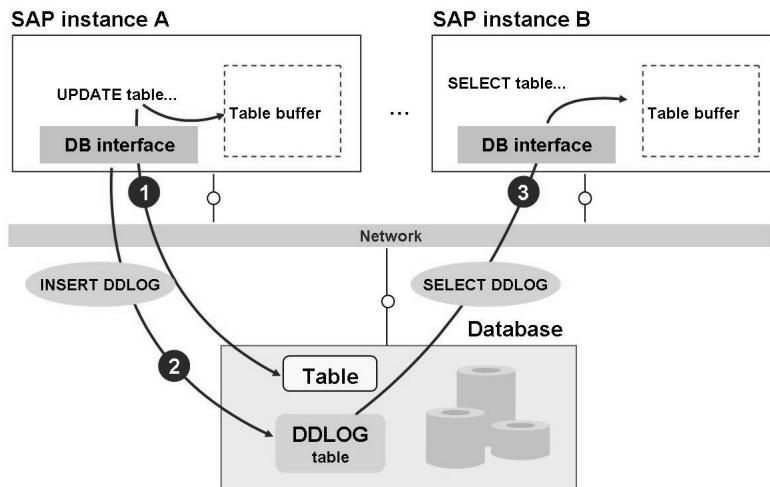


**Hint:** The system parameter for the synchronization interval is *rdisp/bufrefresh*, which specifies the length of the interval in seconds. The value must be between 60 and 3600. A value between 60 and 240 is recommended.

The following diagram illustrates, how buffer synchronization takes place:



**Buffer synchronization of the SAP application servers:  
rdisp/bufreftime = every 1-2 minutes**



**Figure 185: DDLOG Mechanism**

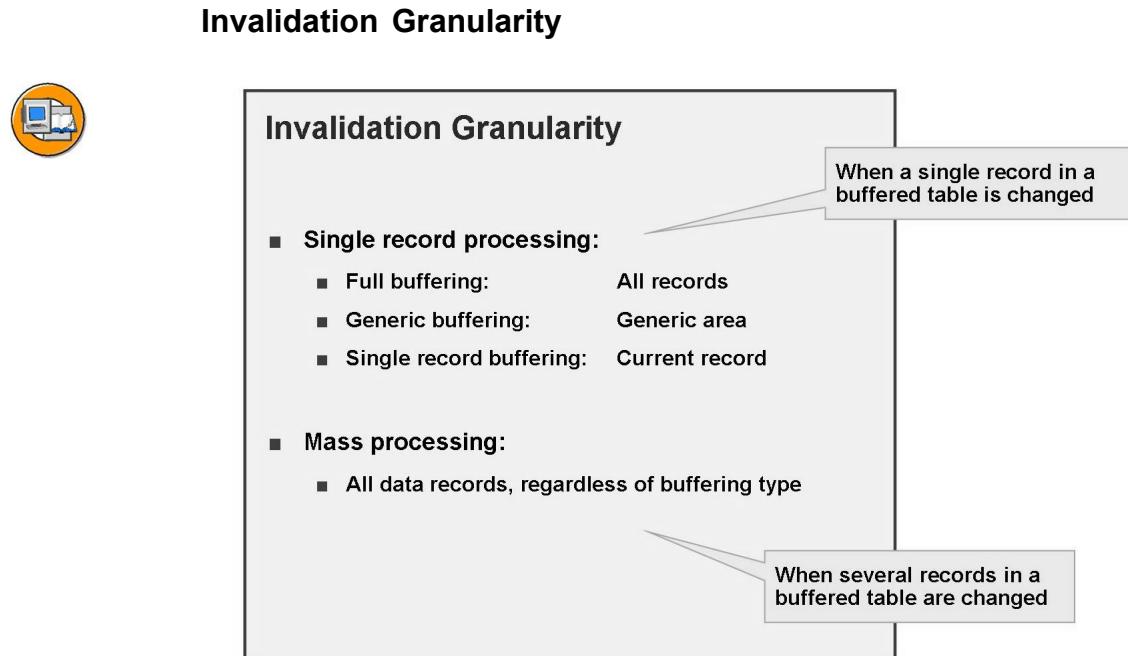
1. In the first step of a database write operation to a buffered table, the corresponding records in the database are changed, and also changed (work area mode) or invalidated (set mode) in the table buffer of the local SAP instance (here: application server A). The table buffers of all other SAP instances (here: application server B) are not changed.
2. In the second step, the local database interface (here: application server A) propagates the changes to the other application servers by writing an appropriate entry to database table DDLOG. The database buffers of the other SAP instances are still not current.
3. In the third step, the database INTERFACES of the SAP instances start buffer synchronization (every 1-2 minutes => controlled by profile parameters). To do so, a SELECT statement is sent to table DDLOG. If the returned records in table DDLOG indicate that change access has been made to buffered tables, the data in the table buffers of the non-local SAP instances is invalidated in accordance with the buffering type. Subsequent SQL statements requesting invalidated data are served by the database.



**Note:** Once data were invalidated in the buffer the system waits for a certain number of read accesses (controlled by a profile parameter) before they are stored in the buffer again. This is done to prevent a fast invalidating and buffering of data.



**Caution:** Since SAP table buffering does not synchronize immediately, but instead once the synchronization interval passes, temporary data inconsistencies can occur. Therefore, you should only buffer data if the reading of obsolete data (within this limit) can be tolerated.



**Figure 186: Invalidating Buffered Records**

SAP buffer contents are invalidated during buffer synchronization. The scope to which data is invalidated depends on the type of access, in addition to the buffering type:

#### Work area mode (single record access)

This means only a single record is changed, deleted, or inserted. This is done with the UPDATE/INSERT/MODIFY/DELETE dbtab (FROM wa) or INSERT dbtab VALUES wa ABAP statements.

#### Set mode (mass access)

This means several data records are changed, deleted, or inserted with a single database access. Database update access as mass processing is formulated in ABAP with the UPDATE/INSERT/MODIFY/DELETE dbtab FROM itab, UPDATE dbtab SET <field> = <value> WHERE <field> = <condition>, or DELETE dbtab WHERE <field> = <condition> statements.

Database update accesses generally invalidate all the records in **fully buffered tables**.

For **generically buffered tables** in work area mode, the records whose generic area has the same expression as the work area fields in the SQL update statement are invalidated. If a generically buffered table is accessed in set mode, all the records are invalidated.

If a **single-record-buffered record** is accessed in work area mode, only the changed single record is invalidated. If an update access is performed in set mode, the entire single-record-buffered table is invalidated.

→ **Note:** Therefore, the granularity of the invalidation corresponds to the granularity used to fill the table buffers.

### Criteria for Table Buffering

The following diagram should help you decide whether to use table buffering or not.



#### When can you buffer tables?

- **Criteria for buffering (rules of thumb):**
  - Small tables, < 10 MB
  - Frequently read, few changes
  - Temporary inconsistency (dirty read) tolerable
  - Accessed primarily using key fields
  - Access using secondary indexes not possible
  - Check memory before buffering additional tables
- **Never buffer transaction data**
  - Too large, too many changes
- **In general, do not buffer master data**
  - Too large, too many access paths
- **In general, buffer Customizing data**
  - Small, few changes

Figure 187: Criteria for Table Buffering

The records of the buffered tables are held redundantly in the shared memory of the participating application servers. As such, you should only buffer tables that are relatively small. Tables that are larger than 10 MB should only be buffered in exceptional cases. To keep the number of invalidations and subsequent buffer load operations as small as possible, you should only buffer tables that are accessed primarily for read access (change accesses < 1% of read accesses). Immediately after a change access to a buffered table, the table buffers of the non-local SAP instances are no longer current (see Buffer Synchronization). Inconsistent data may be read during this period. This must be tolerable. The records of buffered tables are saved sorted by the primary key. Accordingly, you should use key fields to access them. Secondary indexes cannot be used.

Before you decide to buffer tables, you have to consult your system administrator to make sure that sufficient space is available in shared memory (single record buffer; approx. 40 MB; generic table buffer; approx. 80 MB). Otherwise the table

buffers could overwrite each other, which would be counterproductive. Before you change the buffer settings for standard SAP tables, you should search for appropriate Notes in the SAP Notes system. You need a modification key to perform the changes.

## Accessing Buffered Database Tables

If you want to benefit from the SAP table buffering when accessing a buffered database table. Your SELECT statement has to meet certain conditions. Only then data are stored in the buffer or retrieved from it.



### Single record buffering

Buffer only used for SELECT SINGLE ... with fully qualified key

### Generic buffering

Only if first n key fields are restricted to exactly one value

### All buffering types

No additions that have to be processed by the database (see complete list in last chapter)

If these conditions are not met the SELECT statement will bypass the buffer.

→ **Note:** As of *NetWeaver 7.0 EhP2*, it is no longer required to access single record buffered tables with statement SELECT SINGLE. As long as the key is fully qualified in the WHERE condition, the buffer is also used in an array fetch or SELECT loop.



**Hint:** There is a dedicated check in the *Code Inspector* tool that helps you to identify SELECT statements that bypass the SAP table buffer.



## Lesson Summary

You should now be able to:

- Name the relevant system components related to SAP Open SQL
- Describe the process flows involved in a database access using Open SQL
- Explain the following terms: database interface, SAP table buffer, database buffer, cursor cache, database index, and optimizer
- Explain the importance of the SAP database interface

# Lesson: WHERE Condition and Target Area

## Lesson Overview

In this less, you learn which logical expressions can be used in WHERE conditions of Open SQL statements. You learn about a variety of other operators aside from simple value comparisons with “=” and “EQ”.

For specifying data objects in the INTO clause, you have learned so far how to specify structures and internal tables, which must be filled each time. Here, as well, there are alternatives and variants that can be useful in specific situations.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain the importance of the possible operators in WHERE conditions
- Use the different operators correctly
- Know the options available for specifying data objects after INTO
- Implement sequential processing of large data volumes

## Business Example

You want to implement database accesses with Open SQL in your applications. These database accesses must be sophisticated and efficient. Therefore, you want to find out about the special options of using logical expressions in the WHERE condition and specifying data objects in INTO clauses.

## WHERE Conditions

So far, when formulating WHERE conditions, we largely compared with single values and evaluated selection tables (select-options). The comparison values here were data objects (variables and literals). Open SQL supports a variety of other operators, as well as the possibility of comparing database table fields with one another. We examine these kinds of WHERE conditions in this section.

## Operators

The table below shows a list of operators that Open SQL accepts in WHERE conditions.

 **Note:** Some of these operators can have a negative impact on performance if they use potential index fields for delimitation, because the database optimizer cannot find an efficient access strategy.



### Operators in WHERE Conditions

Operator	Meaning and use
=, <, >, <=, <=, <> EQ, LT, GT, LE, GE, NE	Comparison with a single value. In character-type fields, the result of size comparisons may depend on the database code page.
IN (dobj1, dobj2, ...)	Comparison with a list of single values.
BETWEEN dobj1 AND dobj2	Comparison with an interval. In character-type fields, the result may depend on the database code page.
LIKE dobj	Comparison with character strings. The _ and % placeholders let you define a comparison pattern in dobj.
IN seltab	Evaluation of a selection table (select-options)
IS [NOT] NULL	Checks whether the database field has a null value.
AND, OR	Link of logical expressions.
NOT	Negation of a logical expression.

The following diagram shows an example of a complex search for airline customers using name and address data.



#### Example: SELECT with complex WHERE condition

```
DATA lt_customers TYPE TABLE OF scustom.
SELECT * FROM scustom INTO TABLE lt_customers
WHERE country IN ('DE', 'US')
      AND discount BETWEEN '005' AND '010'
      AND postcode LIKE '_ _ _ _ 5_'
      AND name    LIKE '%ra%'.
```

Country selection using list of single values

Range for discount

Postal code has five places and the number "5" in position 4

Name contains substring "ra" in any position

#### Wildcards for LIKE:

\_ = Placeholder for a single character

% = Placeholder for substring of any length

**Figure 188: Operators in WHERE Conditions**

**Hint:** The “\_” and “%” placeholders correspond to the SQL standard in delimitation with LIKE. In other ABAP statements, the “+” and “\*” characters are used for similar comparisons, for example, within comparison tables.

If you want to search for the “\_” or “%” characters themselves, you can use the ESCAPE addition to define an escape symbol. A condition that the column contents must contain a “\_” in any location could then be formulated as follows:

```
LIKE '%!_%' ESCAPE '!'
```

When conditions are linked, the AND operators are evaluated before the OR operators, as usual. You can set parentheses to change this order as needed.

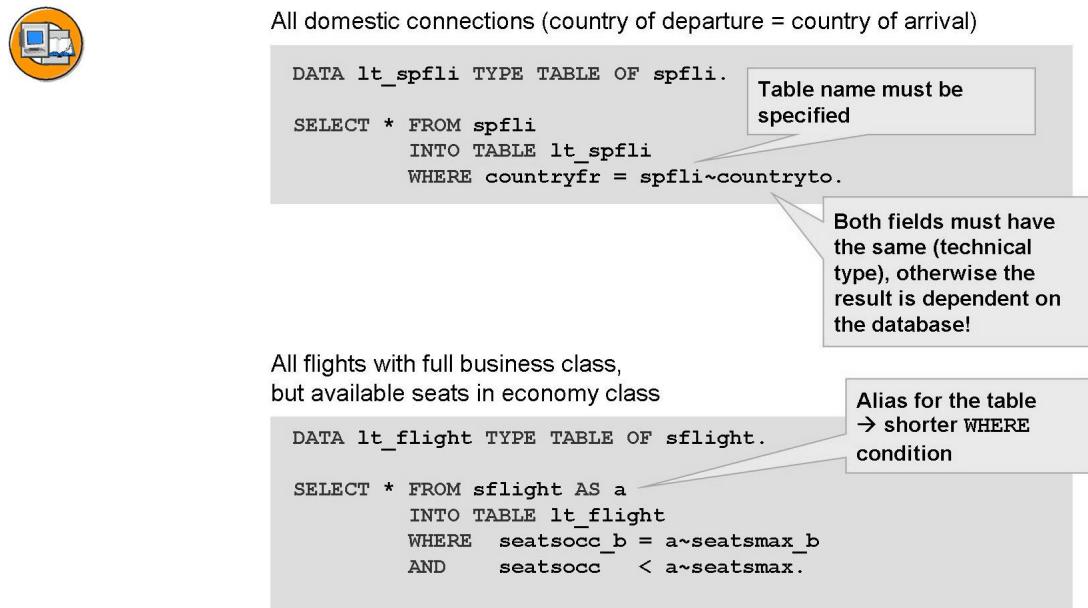


**Note:** The selection of the operator can have a major impact on the usability of a condition for an index search string. The formulation of the WHERE condition influences the search strategy of the optimizer and the performance of the database access.

**Table Field Comparison**

In addition to variables and constants, you can also enter fields from the same database tables after the comparison operators (=, EQ,  $\neq$ , NE, <, LT, and so on) in the WHERE conditions of open SQL statements. This makes it possible to compare two database fields with each other in the WHERE condition and only read the records whose contents fulfill the condition.

The diagram below shows two examples:



**Figure 189: Comparing Table Fields**

To avoid confusing the second table field with a data object, the name of the database table must be specified with the field label connected with the “~” character. Alternatively, you can also use an alias for the table name, which you define with the AS addition after the table name in the FROM clause.

→ **Note:** Aliases for table names play a subordinate role here: They are merely used to abbreviate the WHERE condition. However, we see aliases again in the formulation of joins, where they are crucial if the same database table appears several times in the FROM clause.

⚠ **Caution:** The fields that are compared must have the same basic type and length (ideally, they are based on the same domain). Otherwise the result can depend on how the respective database system stores the different types and how it handles spaces at the end of values.

## Data Objects as Targets of SELECT Statements

When you specify data objects as targets of SELECT statements, you have to differentiate whether the **target area** that is specified after INFO is single line or multiline. Do not confuse this with single line and multiline **result sets**, as shown in the following table:



Statement	Result set	Target area
SELECT SINGLE ...	single-line	single-line
SELECT ... INTO TABLE ...	multiline	multiline
SELECT ... ENDSELECT.	multiline	single-line

The information that can be specified in single line and multiline target areas is discussed below.

### Single Line Target Area

Single line target areas are used for single record access using SELECT SINGLE and in single loops using SELECT ... ENDSELECT. The target is always specified with the INTO addition (without TABLE) or with INTO CORRESPONDING FIELDS OF. The target may be a structure (structured data object). Alternatively, you can also specify a list of elementary data objects or elementary structure components. In this case, the individual fields are separated by commas, while the entire list is enclosed in parentheses.



Filling any list of elementary fields

```
DATA: lv_max TYPE sflight-seatsmax,
      lv_occ TYPE sflight-seatsocc.

SELECT SINGLE seatsmax seatsocc
  FROM sflight INTO (lv_max, lv_occ)
  WHERE ... .
```

Fields comma-separated,  
no spaces behind open  
parenthesis

Targeted filling of individual components of a structure

```
DATA ls_struct TYPE sflight.

SELECT SINGLE seatsmax seatsocc
  FROM sflight
  INTO (ls_struct-seatsmax, ls_struct-seatsocc)
  WHERE ... .

SELECT SINGLE seatsmax seatsocc
  FROM sflight
  INTO CORRESPONDING FIELDS OF ls_struct
  WHERE ... .
```

Explicit components as  
an alternative to  
CORRESPONDING FIELDS  
OF ls\_struct

Figure 190: List of Data Objects Instead of Single Structure



**Hint:** The syntax of this value is only correct if there are **no spaces** between the open parenthesis and the first data object. Any number of spaces can appear before the closing parenthesis and before and after commas, but are not required.

Specifying individual structure components is a more robust alternative to specifying the entire structure after the CORRESPONDING FIELDS OF addition.

In theory, you can also specify a list of individual data objects and structure components after INTO CORRESPONDING FIELDS OF. This is not relevant in practice, however.

### Multiline Target Area

When SELECT statements have multiline target areas, an internal table must always be specified as the data object. You specify the table, for example, after the **INTO TABLE** addition or after **INTO CORRESPONDING FIELDS OF TABLE**.

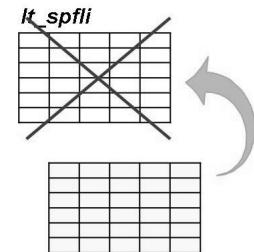


```
DATA lt_spfli TYPE TABLE OF spfli.
```

Filling an internal table (overwrite)

```
SELECT * FROM spfli
  INTO TABLE lt_spfli
  WHERE ... .

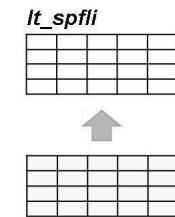
SELECT cityfrom cityto FROM spfli
  INTO CORRESPONDING FIELDS OF
  TABLE lt_spfli
  WHERE ... .
```



Appending lines to an internal table

```
SELECT * FROM spfli
  APPENDING TABLE lt_spfli
  WHERE ... .

SELECT cityfrom cityto FROM spfli
  APPENDING CORRESPONDING FIELDS OF
  TABLE lt_spfli
  WHERE ... .
```



**Figure 191: APPENDING instead of INTO**

Alternatively, you can also replace INTO TABLE and INTO CORRESPONDING FIELDS OF TABLE with **APPENDING TABLE** and **APPENDING CORRESPONDING FIELDS OF TABLE**, respectively. If you use INTO, the contents of the internal table are replaced completely with the selected data; if you use APPEND, the data is added to any existing lines in the table.

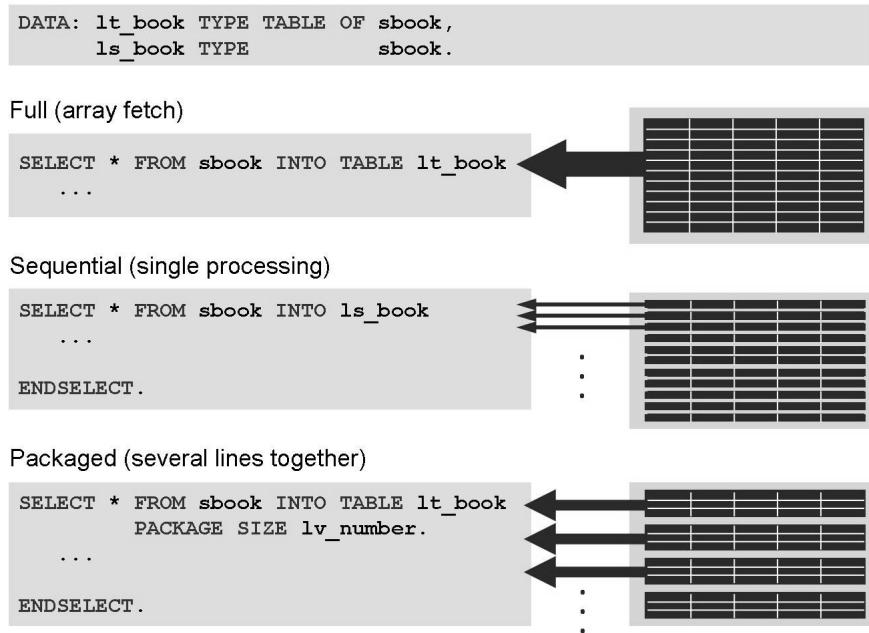


**Hint:** If you use APPENDING for sorted and hashed tables, no “append” in the literal sense is performed. In particular, the addition is not treated in the same way as an index access to the internal table, which means it can also be used for hashed tables. In sorted tables, the new lines are inserted in the existing lines in accordance with the sort sequence.



## Large Volumes of Data

When you process multiline result sets, we recommend reading them into an internal table (array fetch). When huge result sets are involved, however, this risks a program termination, because the available working memory is insufficient for that size of internal table.



**Figure 192: The PACKAGE SIZE Addition**

One solution to this is sequential processing in a SELECT loop. This approach has low memory requirements, as only one record is saved in the program at one time.

→ **Note:** The database interface reads entire packages from the database, of course, but then copies the records to the work area individually. When a package has been processed completely, the next one is requested from the database. As a result, the application server never has to save the entire data volume at once.

A disadvantage of sequential processing in the SELECT loop is the fact that only one record is available to the application log at one time. This is particularly unfavorable when you want to make changes to the records in the database. Since only one record is available at a time, the changes can only be carried out by single record access. However, a large number of single record write accesses is bad for the database and for program performance.

For such (and other) cases, Open SQL provides a variant of the SELECT statement in which the data is processed in a loop (SELECT ... ENDSELECT), but the target area is multiline (INTO TABLE). The number of records that are placed in the

internal table at the same time and processed in the current loop pass is defined with the PACKAGE SIZE addition. You can specify this number either statically or dynamically, but it is defined once at the start of the loop. The same number of lines is then processed during each loop pass.



**Hint:** The value after PACKAGE SIZE determines how many lines the database interface provides to the application program in each loop pass. It has no major impact on the size of the packages that are transferred from the database to the application server.

The sequential processing of packages of different sizes is possible with the following syntax:



Packaged (packages with variable package size)

```
DATA: lt_book    TYPE TABLE OF sbook,
      ls_book    TYPE sbook,
      lv_cursor  TYPE cursor.

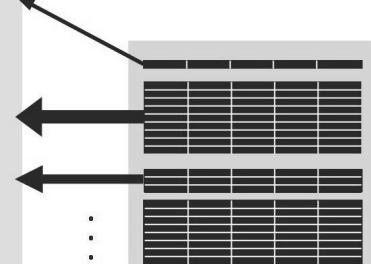
OPEN CURSOR lv_cursor FOR
  SELECT * FROM sbook
  WHERE .....

FETCH NEXT CURSOR lv_cursor
  INTO ls_book.
  ...
DO.
  lv_number = .....
  FETCH NEXT CURSOR lv_cursor
    INTO TABLE lt_book
    PACKAGE SIZE lv_number.
  IF sy-subrc <> 0.
    EXIT.
  ENDIF.
  ...
ENDDO.

CLOSE CURSOR lv_cursor.
```

The cursor variable "remembers" how much of the result set was already processed

The DB access starts here; the data selection (result set) is defined



The DB access ends here and the cursor variable is initialized

Figure 193: OPEN CURSOR ... FETCH ... CLOSE CURSOR

In this variant of database read access, the access itself is detached from the transfer of the result set to the application program.

The **OPEN CURSOR** statement defines and executes the access, but does not pass any data on to the application program.

The **FETCH NEXT CURSOR** statement retrieves the desired number of lines (PACKAGE SIZE addition) from the result set and copies them to an internal table.

The **CLOSE CURSOR** statement completes the database access.

The **cursor variable** - a variable with a special CURSOR type - has a twofold purpose: it identifies the database access in the subsequent statements and also stores the position in the result set of this access up to which processing is already complete. After OPEN CURSOR, the cursor variable points to a position before the first line.

The sequential FETCH statements not only allow you to read different numbers of records; you can also specify a different data object as the target (structures or internal tables) after each INTO.



**Hint:** You can also keep a limit number of accesses open at the same time, for example, to process several database tables in parallel.



**Caution:** The cursor is closed implicitly by a database commit or database rollback. Therefore, no statements that cause an implicit DB commit or DB rollback can appear between OPEN CURSOR and CLOSE CURSOR. When a FETCH statement is performed for a cursor that has already been closed, this raises a catchable exception (exception class CX\_SY\_OPEN\_SQL\_DB).



## Lesson Summary

You should now be able to:

- Explain the importance of the possible operators in WHERE conditions
- Use the different operators correctly
- Know the options available for specifying data objects after INTO
- Implement sequential processing of large data volumes

## Lesson: SELECT Statements: Processing and Aggregating Value Sets

### Lesson Overview

In this lesson, you learn about several additions for the SELECT statement with which you can instruct the database to sort the desired data before it is transferred, aggregate it, or perform calculations.

- **Note:** All the additions and variants of the SELECT statement described in this lesson cause the database interface to ignore any buffering in the SAP table buffer and execute the statement directly in the database. Therefore, the techniques described here are not suitable for buffered tables from a performance perspective. For more information, see the lesson on performance aspects.



### Lesson Objectives

After completing this lesson, you will be able to:

- Request sorted or aggregated data from the database
- Use aggregate functions correctly

### Business Example

You want to implement database accesses with Open SQL in your applications. These database accesses must be sophisticated and efficient. Therefore, you want to find out about the options for sorting and aggregating data, along with the aggregate functions.

### Requesting Sorted Data from the Database

In Open SQL, you can have the database sort the data by specific criteria directly. To do so, you use the ORDER BY addition for the SELECT statement.



Sorting by primary key (ascending):

```
SELECT *
  FROM sflight INTO ... WHERE ...
  ORDER BY PRIMARY KEY.
```

**ORDER BY must appear after the WHERE condition**

```
SELECT mandt carrid connid fldate ...
  FROM sflight INTO ... WHERE ...
  ORDER BY PRIMARY KEY.
```

**All primary key fields (including MANDT) must appear in the field list**

Sorting by any key (ascending):

```
SELECT *
  FROM sflight INTO ... WHERE ...
  ORDER BY connid seatsocc.
```

**Sequence of fields determines priority**

```
SELECT ... connid ... seatsocc ...
  FROM sflight INTO ... WHERE ...
  ORDER BY connid seatsocc.
```

**All sort fields must be in the field list**

Sorting by any key (ascending or descending):

```
SELECT *
  FROM sflight INTO ... WHERE ...
  ORDER BY connid DESCENDING seatsocc ASCENDING.
```

**Figure 194: The ORDER BY Addition - Variants**

In the simplest form, the **ORDER BY PRIMARY KEY** addition, the table is sorted by its complete primary key in ascending order. To use this addition, the following prerequisites must be met:

- The FROM clause must contain a single database table (no views or joins)
- The field list after SELECT must contain all the table's key fields (including the client, which is often forgotten)

The syntax **ORDER BY col1 col2 ...** sorts the data by any existing columns. The sequence of the specified columns determines their sort priority. The following restrictions apply:

- Pooled tables and cluster tables cannot be sorted by simply any field
- You can only sort by columns that appear after the SELECT statement, which means the sort columns must be part of the result set
- You cannot sort by fields with type LCHAR, LRAW, STRING, or RAWSTRING

The following diagram shows an example of the ORDER BY addition with free column selection:



```

TYPES: BEGIN OF lty_s_flightocc,
      connid  TYPE sflight-connid,
      fldate   TYPE sflight-fldate,
      seatsocc TYPE sflight-seatsocc,
   END OF lty_s_flightocc.
DATA lt_flightocc TYPE TABLE OF lty_s_flightocc.

SELECT connid fldate seatsocc
      FROM sflight
      INTO TABLE lt_flightocc
     WHERE seatsocc > 200
      ORDER BY connid DESCENDING seatsmax ASCENDING.

```

Result set sorted first by CONNID, then by SEATSMAX within the CONNID groups

ORDER BY must appear after the WHERE condition

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
001	AA	0018	20081203	143
001	AA	0018	20081207	155
001	AA	0018	20090106	165
001	AA	0019	20081205	244
001	AA	0019	20090114	312

*lt\_flightocc*

CONNID	FLDATE	SEATSOCC
0019	20081205	244
0019	20090114	312
0017	20081230	243
0017	20090102	250
0017	20090106	300
0017	20090119	300
0014	20090121	243
0014	20081217	250

Figure 195: The ORDER BY Addition

If you do not use an addition, the system sorts in ascending order. You can use the optional DESCENDING and ASCENDING additions after a given field to define the sorting direction.

→ **Note:** Sorting can be a expensive process for the database if the result set is large and the sort fields do not agree with the fields in the database index used.

## Requesting Aggregated Data from the Database

The SELECT DISTINCT statement in Open SQL instructs the database to aggregate the result set of a SELECT statement such that it does not contain any duplicate entries. All fields of the result set are used for the comparison. You can either load the result of SELECT DISTINCT into an internal table (array fetch) or process it sequentially (SELECT loop).

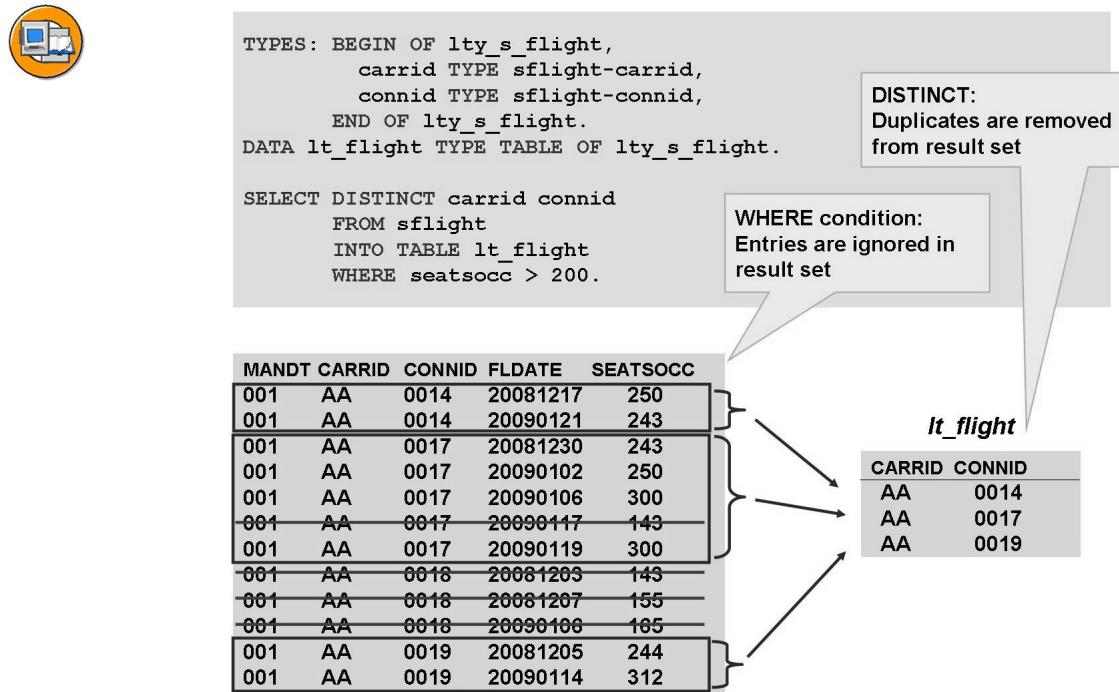


Figure 196: The SELECT DISTINCT Statement

The following restrictions apply to SELECT DISTINCT:

- The field list cannot contain any columns with type LCHAR, LRAW, STRING, or RAWSTRING
- You have to use SELECT DISTINCT \* to access pooled tables and cluster tables; you cannot select individual columns

## Aggregate Expressions

An **aggregate expression** uses an **aggregate function** to specify a column in the SELECT statement. Aggregate determine values from multiple lines in a column of a database table. The calculation is performed in the database system. ABAP Open SQL supports the following aggregate functions:



### Aggregate Functions

Function	Meaning of Result and Conditions	Data Type of Result
MIN( col )	Minimum value in the col column within the result set	Like the col column
MAX( col )	Maximum value in the col column within the result set	Like the col column

Function	Meaning of Result and Conditions	Data Type of Result
SUM( col )	Sum of the contents of the col column in the result set - The col column must be numeric	Like the col column
AVG( col )	Average value of the contents of the col column in the result set - The col column must be numeric	Float (F)
COUNT( * ) COUNT(*)	Number of lines in the result set	Integer (I)

The data object in the INTO clause must provide a structure component or table column with the suitable type (see table) for each aggregate expression after SELECT.

When using aggregate functions, there is a major difference between cases in which the field list consists exclusively of aggregate expressions and cases in which it also contains field labels that are not arguments of an aggregate function.

### Field List Only with Aggregate Expressions

If the field list in a SELECT statement contains only aggregate expressions, the result is a single dataset. A structure may be specified as the target (similar to the SELECT SINGLE statement). The following diagram shows an example:



```

TYPES: BEGIN OF lty_s_flightocc,
        cntall TYPE i,
        minocc TYPE sflight-seatsocc,
        maxocc TYPE sflight-seatsocc,
        sumocc TYPE sflight-seatsocc,
      END OF lty_s_flightocc.
DATA ls_flightocc TYPE lty_s_flightocc.

SELECT COUNT(*) MIN( seatsocc ) MAX( seatsocc ) SUM( seatsocc )
  FROM sflight
  INTO ls_flightocc.

```

Field list only contains  
aggregate functions  
→ One-line result

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
001	AA	0018	20081203	143
001	AA	0018	20081207	155
001	AA	0018	20090106	165
001	AA	0019	20081205	244
001	AA	0019	20090114	312

ls\_flightocc

CNTALL	MINOCC	MAXOCC	SUMOCC
12	143	312	2748

**Figure 197: SELECT Statement with Aggregate Expressions**



**Hint:** If the field list only contains the MIN, MAX, SUM, and AVG functions, but not COUNT, an internal table can also be specified as the target. Since the result is single-line, only the first line of the internal table is filled.

The count function has a second variant in addition to the COUNT(\*) form we have seen so far, in which the argument is a field name with the DISTINCT addition - for example **COUNT( DISTINCT col )**. This aggregate expression returns the number of different (distinct) values in the corresponding column.

You can use the DISTINCT addition for all other aggregate functions as well, for example, **AVG( DISTINCT col )**. The effect here is that values that occur multiple times for the column in the selection result are only included once in aggregate formation. To an extent, the table is aggregated before the aggregate is formed.



```
TYPES: BEGIN OF lty_s_flightocc,
         cntall TYPE i,
         cntcon TYPE i,
         sumocc TYPE sflight-seatsocc,
      END OF lty_s_flightocc.
DATA ls_flightocc TYPE lty_s_flightocc.

SELECT COUNT(*) COUNT( DISTINCT connid ) SUM( DISTINCT seatsocc )
      FROM sflight
      INTO ls_flightocc.
```

Identical values are only taken into account once during totaling

Number of distinct CONNID values

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	1	0014	250
001	AA	1	0014	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
001	AA	0018	20081203	143
001	AA	3	20081207	155
001	AA	0018	20090106	165
001	AA	0019	20081205	244
001	AA	0019	20090114	312

*ls\_flightocc*

CNTALL	CNTCON	SUMOCC
12	4	1812

**Figure 198: Aggregate Functions with the DISTINCT Addition**



**Note:** The aggregate **SUM( DISTINCT seatsocc )** in the example is not informative; it is only intended as an illustration, not as a prime example of using the technique. In general, the DISTINCT addition only plays a subordinate role for the MIN, MAX, SUM, and AVG functions.

When SELECT statements consist only of aggregate expressions, there are several things you have to watch out for:



#### Special case 1: Aggregate functions only

```
SELECT COUNT(*) MIN( seatsmax ) MAX( seatsmax ) SUM( seatsmax )
FROM sflight
INTO ls_flighttocc
WHERE carrid = 'XX'.
```

DB table has no flights with CARRID = 'XX'



*ls\_flighttocc*

CNTALL	MINOCC	MAXOCC	SUMOCC
0	0	0	0

The access still returns a result (initial values) and  
sy-subrc = 0, sy-dbcnt = 1

#### Special case 2: Only aggregate function COUNT(\*)

```
SELECT COUNT(*)
FROM sflight
WHERE carrid = 'AA'
AND connid = '0017'
AND fldate = '20090101'.
IF sy-subrc = 0.
.
ENDIF.
```

Field list only contains aggregate function COUNT(\*) → INTO clause can be omitted

sy-subrc = 4 if no data is found  
sy-subrc = 0 if at least 1 record fulfills the conditions

**Figure 199: Special Situations**

#### Special case 1: The value set is empty before aggregate formation

Unlike SELECT statements, these database accesses return a result even if no matching records are found in the database. In this case, the result of COUNT is zero and the other aggregate functions return initial values. sy-subrc is set to 0 and sy-dbcnt to 1, except in special case 2 below.



**Caution:** If the MIN( col ) aggregate function returns an initial value, for example, you cannot conclude that the table actually contains at least one line with the initial value in the col column. It could also mean that the table does not contain any entries that satisfy the WHERE condition.

#### Special case 2: The SELECT statement only contains the COUNT(\*) function

If the field list only contains the COUNT(\*) aggregate function, you can omit the INTO clause completely. If no data is found in the database, sy-subrc is set to 4 and sy-dbcnt to 0.



**Caution:** You should not use this statement to do existence checks. Use SELECT ... UP TO 1 ROWS WHERE ... instead.

While COUNT(\*) looks for all records that match the WHERE condition, ... UP TO 1 ROWS stops searching after the first record is found.

## Field List with Aggregate Expressions and Field Labels

If the field list of a SELECT contains field labels in addition to aggregate expressions, the result is always **multiline**. As a result, you have to load it into an internal table (array fetch) or process it sequentially (SELECT loop). The SELECT statement must contain the **GROUP BY** addition and all fields that are not arguments of an aggregate function in the field list must be listed after the GROUP BY addition.

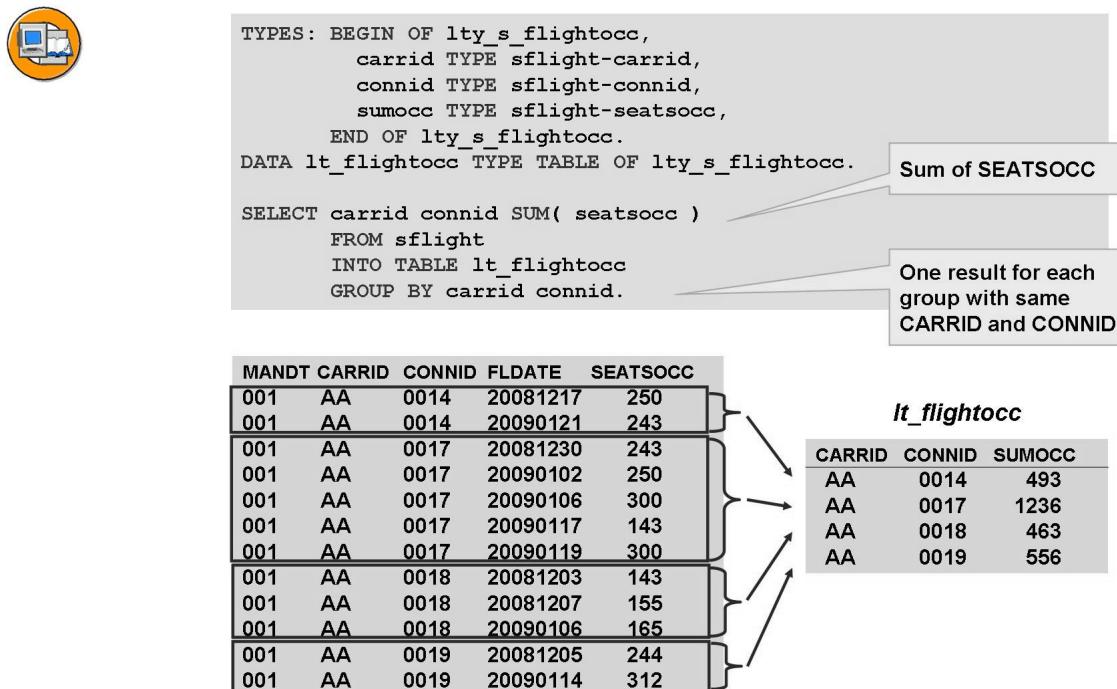


Figure 200: The GROUP BY Addition

When the GROUP BY addition is specified, the database does not apply the aggregate functions to all the found records together, but instead sorts them into groups first. A group contains all the records that have the same contents in the columns specified after GROUP BY. The aggregate functions are evaluated separately for each of these groups. Each group then corresponds to one line in the selection result.

**Note:** All the fields in the field list that are not arguments of an aggregate function must be listed after GROUP BY. GROUP BY can also contain field labels that are not part of the field list – at least theoretically. However, this makes it difficult to trace which line in the result belongs to a specific group. Therefore, every field after GROUP BY should also appear as a column in the selection result.



The following restrictions apply to the GROUP BY addition:

- Individual columns are listed after SELECT (SELECT \* is not allowed)
- No pooled tables or cluster tables
- The fields after GROUP BY cannot have type STRING or RAWSTRING

When you use GROUP BY, you can specify a logical expression after the HAVING expression to restrict the result set further.

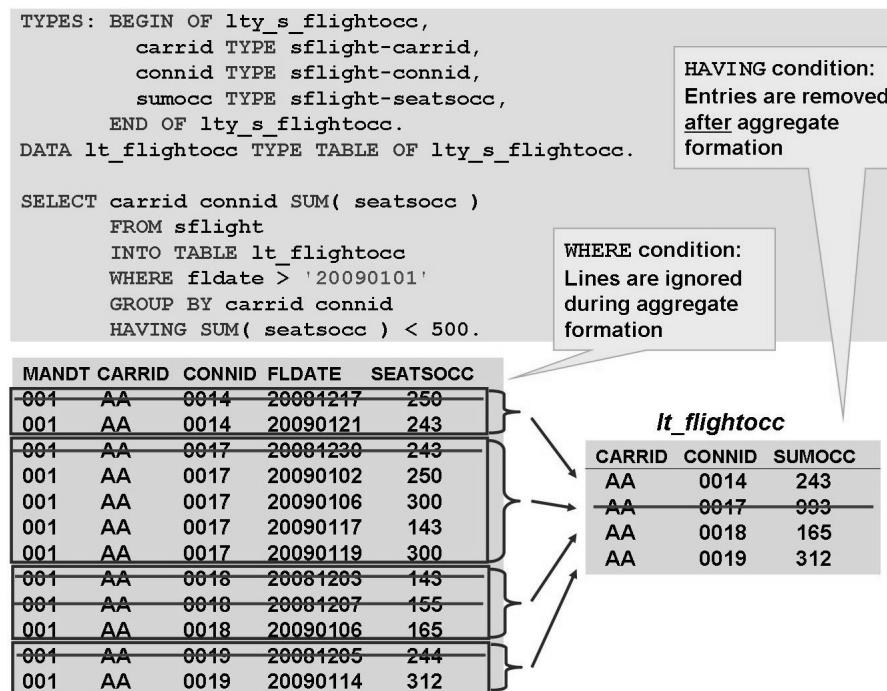


Figure 201: The HAVING Addition

In contrast to the WHERE condition, the logical expression after HAVING can also contain aggregate functions (in SAP Web AS 6.10 and later). Aside from the aggregate functions, the logical expression after HAVING can only contain fields that are specified after GROUP BY.



**Caution:** If you use field labels after HAVING that are not listed after GROUP BY, this does not cause a syntax error, but results in a catchable runtime error (CX\_SY\_OPEN\_SQL\_DB).

The aggregate functions after HAVING can be different from the aggregate functions after SELECT. In the above example, you could specify a condition COUNT(\*) > 1 after HAVING to only evaluate groups that contain more than one flight; you do not have to use the aggregate COUNT(\*) function after SELECT in this case.

# Exercise 18: SELECT Statements – Aggregating Value Sets

## Exercise Objectives

After completing this exercise, you will be able to:

- Use the DISTINCT addition for SELECT statements
- Request sorted data from the database
- Use aggregate functions correctly

## Business Example

You want to enhance a program that outputs a customer's flight bookings. Based on the list of bookings, you also want to output the total bookings by currency and an alphabetical list of the involved travel agencies.

### Template:

BC402\_DBT\_SQL\_CONDENSE

### Solution:

BC402\_DBS\_SQL\_CONDENSE

### Task 1:

Copy the template BC402\_DBT\_SQL\_CONDENSE to the name **ZBC402\_##\_CONDENSE**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Which input options are available on the selection screen?

---

---

---

3. Which data is output in the list? Which data do you need, but is still missing?

---

---

---

## Task 2:

Implement subroutine GET\_TRAVELAGS to determine the travel agencies that made the bookings. Make sure of the following:

Each travel agency only appears once

The travel agencies are sorted alphabetically by name

Implement a single database access that returns the data completely in the desired form. Read from database view BC402\_SCUS\_BOOK.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which component contains the line type of ct\_travelags?

---

---

---

2. Implement a SELECT statement and fill ct\_travelags with the data for the travel agencies. Access database view BC402\_SCUS\_BOOK and limit the access with the customer number.
3. Supplement the SELECT statement with an addition that instructs the database to remove repeated entries from the result set.
4. Supplement the SELECT statement with an addition that instructs the database to return the result set sorted by travel agency name.

*Continued on next page*

### Task 3:

Implement subroutine GET\_SUMS to calculate the booking totals separated by currency.

Program a single database access that returns the data completely in the desired form. Read directly from database table SBOOK.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which component does the line type of ct\_sums contain?

---

---

---

2. Implement a SELECT statement and fill ct\_sums with the booking totals. Access database table SBOOK and limit the access with the customer number.

Use aggregate functions to have the database calculate the totals.

3. Which addition can you use to calculate the totals separated by currency?

---

---

---

## Solution 18: SELECT Statements – Aggregating Value Sets

### Task 1:

Copy the template BC402\_DBT\_SQL\_CONDENSE to the name **ZBC402\_##\_CONDENSE**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Which input options are available on the selection screen?  
**Answer:** An airline customer number
3. Which data is output in the list? Which data do you need, but is still missing?  
**Answer:** Data output:
  - Customer's name and form of address
  - A list of all the customer's bookings (without cancellations)Still missing:
  - A list of total bookings (separated by currency)
  - A list of travel agencies who made the bookings

### Task 2:

Implement subroutine GET\_TRAVELAGS to determine the travel agencies that made the bookings. Make sure of the following:

- Each travel agency only appears once
- The travel agencies are sorted alphabetically by name

Implement a single database access that returns the data completely in the desired form. Read from database view BC402\_SCUS\_BOOK.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which component contains the line type of ct\_travelags?

**Answer:**

- AGENCYNUM:** Travel agency number
- NAME:** Travel agency name
- CITY:** City

*Continued on next page*

2. Implement a SELECT statement and fill ct\_travelags with the data for the travel agencies. Access database view BC402\_SCUS\_BOOK and limit the access with the customer number.
  - a) See the source code excerpt from the model solution.
3. Supplement the SELECT statement with an addition that instructs the database to remove repeated entries from the result set.
  - a) DISTINCT addition. See the source code excerpt from the model solution.
4. Supplement the SELECT statement with an addition that instructs the database to return the result set sorted by travel agency name.
  - a) ORDER BY addition. See the source code excerpt from the model solution.

### Task 3:

Implement subroutine GET\_SUMS to calculate the booking totals separated by currency.

Program a single database access that returns the data completely in the desired form. Read directly from database table SBOOK.

1. Analyze the interface of the subroutine, particularly the typing of the interface parameters. Which component does the line type of ct\_sums contain?

**Answer:**

**FORCURAM:** Price of booking in foreign currency (dependent on booking location)

**FORCURKEY:** Payment currency

2. Implement a SELECT statement and fill ct\_sums with the booking totals. Access database table SBOOK and limit the access with the customer number.

Use aggregate functions to have the database calculate the totals.

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

3. Which addition can you use to calculate the totals separated by currency?

**Answer:** GROUP BY forcurkey.

### Result

Source code excerpt from the model solution:

```
REPORT  bc402_dbs_sql_condense.
```

*Continued on next page*

```

TYPES: BEGIN OF gty_s_sums,
       forcuram TYPE sbook-forcuram,
       forcurkey TYPE sbook-forcurkey,
   END OF gty_s_sums.

TYPES: BEGIN OF gty_s_travelags,
       agencynum TYPE bc402_scus_book-agencynum,
       name      TYPE bc402_scus_book-name,
       city      TYPE bc402_scus_book-city,
   END OF gty_s_travelags.

TYPES:
  gty_t_sums      TYPE SORTED TABLE OF gty_s_sums
                  WITH UNIQUE KEY forcurkey,

  gty_t_bookings  TYPE STANDARD TABLE OF scus_book
                  WITH NON-UNIQUE KEY
                  carrid connid fldate bookid,

  gty_t_travelags TYPE STANDARD TABLE OF gty_s_travelags
                  WITH NON-UNIQUE KEY agencynum.

DATA:
  gs_customer    TYPE scustom,
  gt_bookings    TYPE gty_t_bookings,
  gt_sums        TYPE gty_t_sums,
  gt_travelags   TYPE gty_t_travelags.

PARAMETERS:
  pa_cust TYPE sbook-customid DEFAULT '00000001'.

START-OF-SELECTION.

SELECT SINGLE * FROM scustom INTO gs_customer
  WHERE id = pa_cust.

SELECT * FROM scus_book
  INTO TABLE gt_bookings
  WHERE customid = pa_cust
  AND cancelled <> 'X'.

PERFORM get_sums USING     pa_cust
                  CHANGING gt_sums.

```

*Continued on next page*

```

        PERFORM get_travelags USING      pa_cust
                           CHANGING gt_travelags.

        PERFORM output_list USING gs_customer
                           gt_bookings
                           gt_sums
                           gt_travelags.

*&-----*
*&      Form  GET_SUMS
*&-----*
FORM get_sums  USING      pv_customid TYPE sbook-customid
                  CHANGING ct_sums      TYPE gty_t_sums.

        SELECT SUM( forcuram ) forcurkey
              FROM sbook
              INTO TABLE ct_sums
              WHERE customid = pv_customid
                GROUP BY forcurkey.

ENDFORM.          " GET_SUMS

*&-----*
*&      Form  GET_TRAVELLAGS
*&-----*
FORM get_travelags  USING      pv_customid TYPE sbook-customid
                      CHANGING ct_travelags TYPE gty_t_travelags.

        SELECT DISTINCT agencynum name city
              FROM bc402_scus_book
              INTO TABLE ct_travelags
              WHERE customid = pv_customid
                ORDER BY name.

ENDFORM.          " GET_TRAVELLAGS

*&-----*
*&      Form  OUTPUT_LIST
*&-----*
FORM output_list  USING ps_customer  TYPE scustom
                  pt_bookings  TYPE gty_t_bookings
                  pt_sums      TYPE gty_t_sums
                  pt_travelags TYPE gty_t_travelags.

```

*Continued on next page*

```

DATA:
  ls_bookings  LIKE LINE OF pt_bookings,
  ls_sums      LIKE LINE OF pt_sums,
  ls_travelags LIKE LINE OF pt_travelags.

DATA lv_text TYPE string.

CONCATENATE text-wcm
  ps_customer-form
  ps_customer-name
  INTO lv_text
  SEPARATED BY space.
CONDENSE lv_text.

WRITE: / lv_text.
SKIP.

WRITE / text-lob.
ULINE.
SKIP.

LOOP AT pt_bookings INTO ls_bookings.
  WRITE: /
    ls_bookings-bookid,
    ls_bookings-carrid RIGHT-JUSTIFIED,
    ls_bookings-connid ,
    ls_bookings-fldate,
    ls_bookings-cityfrom RIGHT-JUSTIFIED,
    '->',
    ls_bookings-cityto.
ENDLOOP.
SKIP 2.

WRITE: / text-sum.
ULINE.
SKIP.

LOOP AT pt_sums INTO ls_sums.
  WRITE: /
    ls_sums-forcuram CURRENCY ls_sums-forcurkey,
    ls_sums-forcurkey.
ENDLOOP.
SKIP 2.

```

*Continued on next page*

```
WRITE / text-tag.  
ULINE.  
SKIP.  
  
LOOP AT pt_travelags INTO ls_travelags.  
  WRITE: /  
    ls_travelags-name,  
    ls_travelags-city.  
  ENDLOOP.  
  
ENDFORM.          " OUTPUT_LIST
```



## Lesson Summary

You should now be able to:

- Request sorted or aggregated data from the database
- Use aggregate functions correctly

# Lesson: Reading from Multiple Database Tables

## Lesson Overview

Below you find various techniques that can be used to read data from several tables.



## Lesson Objectives

After completing this lesson, you will be able to:

- Use views and formulate joins correctly
- Understand and use subqueries
- Use the FOR ALL ENTRIES addition correctly

## Business Example

You want to implement database accesses with Open SQL in your applications. These database accesses must be sophisticated and efficient. Cases in which data has to be read from several databases in combination are especially critical. Therefore, you want to find out about the techniques you can use to improve performance when accessing multiple database tables.

## Overview

When data is stored in a relational database, it is normal that when data is accessed in one database table (the primary table), entries from other database tables (secondary tables) have to be read as well.

One obvious method accesses the secondary tables in a loop over the records from the primary table. This approach is called **nested SELECTs**.



```

SELECT * FROM t1 WHERE ...
SELECT * FROM t2 WHERE ...
SELECT * FROM t3 WHERE ...
SELECT * FROM t4 WHERE ...
SELECT * FROM t5 WHERE ...
...
ENDSELECT.
ENDSELECT.
SELECT * FROM t6 WHERE ...
SELECT * FROM t7 WHERE ...
SELECT * FROM t8 WHERE ...
...
ENDSELECT.
ENDSELECT.
ENDSELECT.
ENDSELECT.

```

**!**

*Results in:*

- Lots of data packages
- Lots of identical accesses
- A large transfer effort

**Figure 202: Nested SELECTs**

Since it may be necessary to read additional data for the data in the secondary tables, and so on, there is no limit to the nesting depth.

For a number of reasons, nested SELECTS are the most memory-intensive method for reading multiple database tables and should be avoided whenever possible. They cause a high load on the database and on the network, because many partially filled data packages are transferred and the same data may be read several times in a row (identical accesses).

In this lesson, you learn several techniques that let you avoid nested SELECTS completely, or at least reduce their negative impact on system performance.

## ABAP Join and Database Views

To read data that is distributed across multiple tables, you have to create a link between the functionally dependent tables. The corresponding logical database operator is called a JOIN.

You can use either **database views** in the *ABAP Dictionary* or **ABAP joins** to implement a JOIN.



**Hint:** Database views and ABAP joins can only be used with **transparent tables**. If you are working with **pooled tables or cluster tables**, you have to use the techniques described further below.

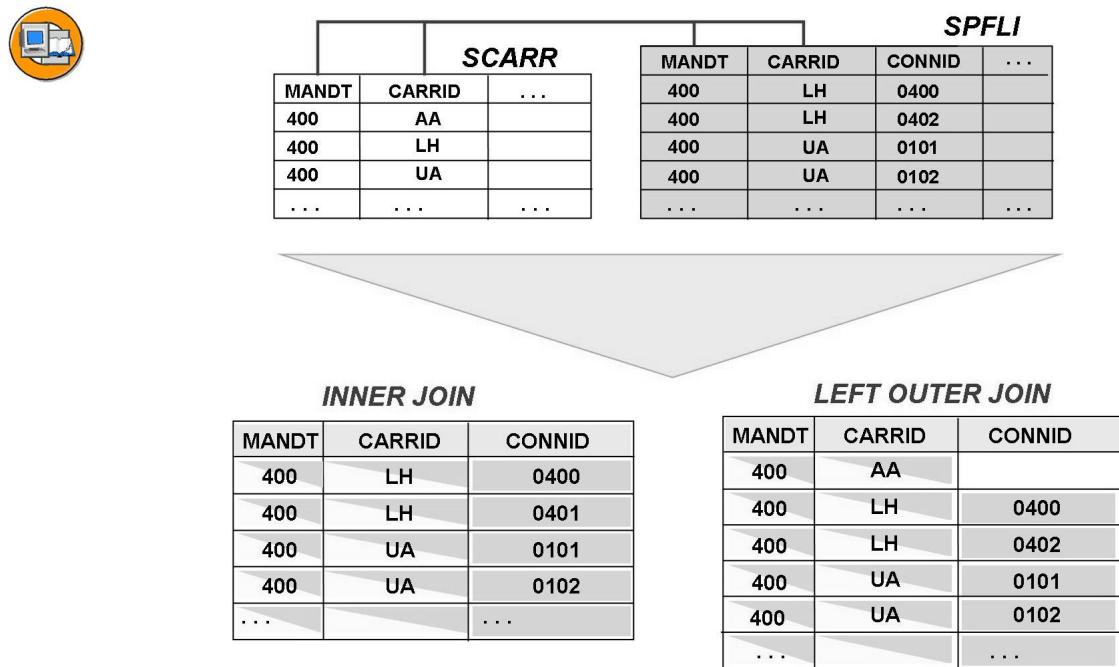


Figure 203: Example: Inner/Outer Join

You can derive the logic of the INNER JOIN and OUTER JOIN from the intended result set.

An **INNER JOIN** corresponds to the result set that only considers the records from the outer table for which suitable data records exist in the inner table (on the left in the above example).

A **LEFT OUTER JOIN** corresponds to the result set that contains all the records from the outer table, regardless of whether or not suitable records exist in the inner table. If no suitable records exist in the inner table, the fields of the inner table are set to ZERO values in the result set. The tables involved in a JOIN are called base tables. The result of a join can be a projection (column selection) or a selection (line selection).

Since not all of the databases supported by SAP support the standard syntax for ON conditions, restricted syntax must be used to ensure that only JOINS that return the same result set on all database systems are allowed.

### Restrictions for Outer Join

- You can only have a table or a view to the right of the JOIN operator; you cannot have another JOIN expression
- Only AND can be used as a logical operator in an ON condition
- Every comparison in the ON condition must contain a field from the table on the right
- None of the fields in the table on the right can appear in the WHERE conditions of the LEFT OUTER JOIN
- For more information about implementing JOIN functions, see the ABAP documentation.



### Example: ABAP INNER Join

```
SELECT <fieldlist> INTO <target>
  FROM <dbtab1> [AS <alias1>]
    INNER JOIN <dbtab2> [AS <alias2>]
      ON <alias1>~<dbtab1-field1> = <alias2>~<dbtab2-field1>
        AND <alias1>~<dbtab1-field2> = <alias2>~<dbtab2-field2>
          AND ...
            WHERE ...
...
ENDSELECT.
```

```
REPORT zselect_view.

SELECT f~carrid b~connid ... INTO ( ... )
  FROM scarr AS f INNER JOIN spfli AS b
    ON f~carrid = b~carrid
      WHERE ...
...
ENDSELECT.
```

**Figure 204: Example: ABAP Inner Join**

A disadvantage of using ABAP joins is that the statement is more complex than a Dictionary view, where the syntax of the SELECT statement (specifically, the FROM clause) corresponds to a regular table access.



**Hint:** Note that ABAP joins always bypass the table buffer, which of course is only a disadvantage for joins involving buffered tables. Dictionary views can be buffered in *SAP R/3 Release 4.0* and later (depending on which tables are used in the view).

### Attributes and Benefits of Database Views

- You can use views in other programs as well.
- There are where-used lists and search functions (SE84 / SE81) to find existing views quickly.
- Like database tables, you can buffer views (technical settings).
- Fields common to both tables (join fields) are only transferred from the database to the application server once.
- The view is implemented in the ABAP Dictionary as an inner join. This means no data is transferred if the inner table does not contain any entries that correspond to the outer table.
- If you do not want to use an inner join to read from a text table, for example (which cannot deal with situations in which the join results do not contain any records, because no entry is available in a certain language), use an ABAP left outer join (see Left Outer Join).



**Hint:** A disadvantage of both ABAP joins and database views is that **redundant** data from the outer table appears in the result set if there is a 1:n relationship between the outer and inner tables. This can considerably increase the amount of data transferred from the database. **For this reason, you should use a field list with a join to identify only those fields that you really need.** The runtime of a join formation is highly dependent on the database optimizer, especially when the join involves more than two tables. However, it is usually faster than using nested SELECT statements.

### Special Techniques

It is not always possible to avoid nested SELECTs with ABAP joins or database views - especially when you are using pooled tables and clustered tables, which do not support these techniques.

You may find the following alternatives useful, depending on the situation at hand:

## SUBSELECT and SUBQUERY



Subquery returns single value

```
SELECT * FROM sflight
  INTO TABLE lt_flights
 WHERE seatsocc =
      ( SELECT MAX( seatsocc ) FROM sflight ).
```

Select all flights in SFLIGHT  
with a maximum of  
passengers

Subquery returns single-column, multiline result

```
SELECT * FROM scarr
  INTO TABLE lt_carriers
 WHERE carrid IN
      ( SELECT DISTINCT carrid FROM spfli
        WHERE cityfrom = 'FRANKFURT' ).
```

Select airlines with  
departure city "Frankfurt"

Subquery returns any result

```
SELECT * FROM scarr
  INTO TABLE lt_carriers
 WHERE EXISTS ( SELECT carrid FROM spfli
                  WHERE carrid = scarr~carrid
                    AND cityfrom = 'FRANKFURT' ).
```

Select airlines with  
departure city "Frankfurt"

**Figure 205: Subselects / Subqueries**

A subquery is a query within a SELECT, UPDATE, or DELETE statement. It is formulated in the WHERE or HAVING clause to check whether the data in various database tables or views possess certain attributes.

A SELECT statement with a subquery has a more restricted syntax than a SELECT statement without a subquery: `SELECT ... FROM dbtab [WHERE ...] [GROUP BY ...] [HAVING ...]`.

If the subquery returns **exactly one value**, the usual comparison operators aside from LIKE and BETWEEN can be used. If a subquery is used with a comparison operator instead of with EXISTS, then the SELECT clause of the subquery can only contain a single column, which can be a field in the database table or an aggregate expression. In the second example above, the subquery is supposed to return several lines, each with one value. If you want to compare all the returned values, you have to use IN. Subqueries whose WHERE condition contains fields from the main query (third example) are called correlated subqueries. If subqueries are nested, each subquery can use all the fields from the higher-level subqueries in

the hierarchy. You should also formulate positive subqueries whenever possible. Negative formulations may result in performance-intensive database analyses if no adequate index is available.



**Hint:** In many cases, you can also use a join to obtain the desired result of a subquery; the join is easier to read. No blanket statements can be made about the performance of subqueries. If you want to use subqueries, be sure to test them comprehensively beforehand.

## Parallel Cursors

In a previous lesson, you saw how you can use the Open SQL statements OPEN CURSOR, FETCH NEXT CURSOR, and CLOSE CURSOR to program explicit cursor handling. You also saw how cursors can be open in several database tables at the same time. This lets you process the contents of multiple database tables in parallel without using nested SELECTs.

This is illustrated in the following example:

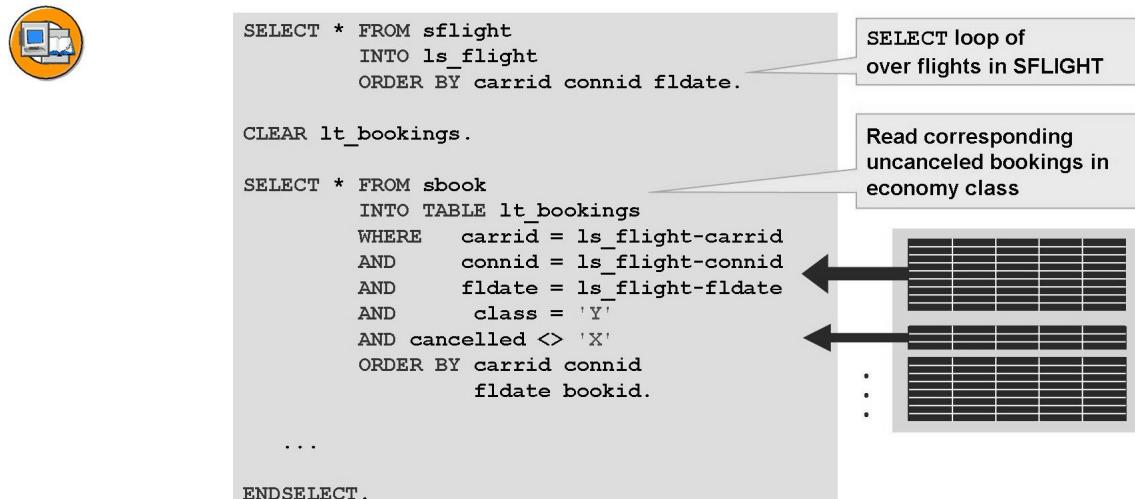


Figure 206: Initial Situation: Nested SELECTs

You want the program to process all flights from database table SFLIGHT. You want to read the non-cancelled bookings in the economy class for each flight from database table SBOOK and process the bookings together with the flight. The obvious solution, although it may have poor performance, is a nested SELECT.

Due to the large data volume involved, we do not want to buffer the bookings completely in an internal table.

The following diagram shows how to achieve the same result by addressing the two database tables with two concurrently open database cursors:



```

OPEN CURSOR lv_cursor_sflight FOR
  SELECT * FROM sflight
  ORDER BY carrid connid fldate.

OPEN CURSOR lv_cursor_sbook FOR
  SELECT * FROM sbook
  WHERE class = 'Y' AND cancelled <> 'X'
  ORDER BY carrid connid fldate bookid.

DO.
  FETCH NEXT CURSOR lv_cursor_sflight
    INTO ls_flight.
  IF sy-subrc <> 0.
    EXIT.
  ENDIF.

  CLEAR lt_bookings.
  IF ls_flight-seatsocc > 0.
    FETCH NEXT CURSOR lv_cursor_sbook
      INTO TABLE lt_bookings
      PACKAGE SIZE ls_flight-seatsocc.
  ENDIF.
  ...
ENDDO.

CLOSE CURSOR lv_cursor_sbook.
CLOSE CURSOR lv_cursor_sflight.

```

**Two cursors for**

- All flights
- Uncanceled bookings in economy class

**seastocc = Number of uncanceled bookings in economy class**  
→ The next seatsocc bookings are are bookings for the current flight

**Figure 207: Parallel Cursors instead of Nested SELECTs**

Each fetch for the first cursor (cursor variable lv\_cursor\_sflight) places the next respective record from database table SFLIGHT into structure ls\_flight (similar to a SELECT loop).

Each fetch for the second cursor (cursor variable lv\_cursor\_sbook) reads the corresponding non-cancelled economy bookings into internal table lt\_bookings. These bookings are not actually selected based on their key, however; instead, the next n records are simply read. The value of n corresponds to the value of the seatsocc field in the record for the flight. Since the selection results are sorted (ORDER BY addition), the next n bookings are the bookings for the current flight.

The following general prerequisites must be met to use this technique:

- Both tables must be sorted by the same criteria (ORDER BY addition)
- Every record in the outer table must specify exactly how many corresponding records exist in the inner table

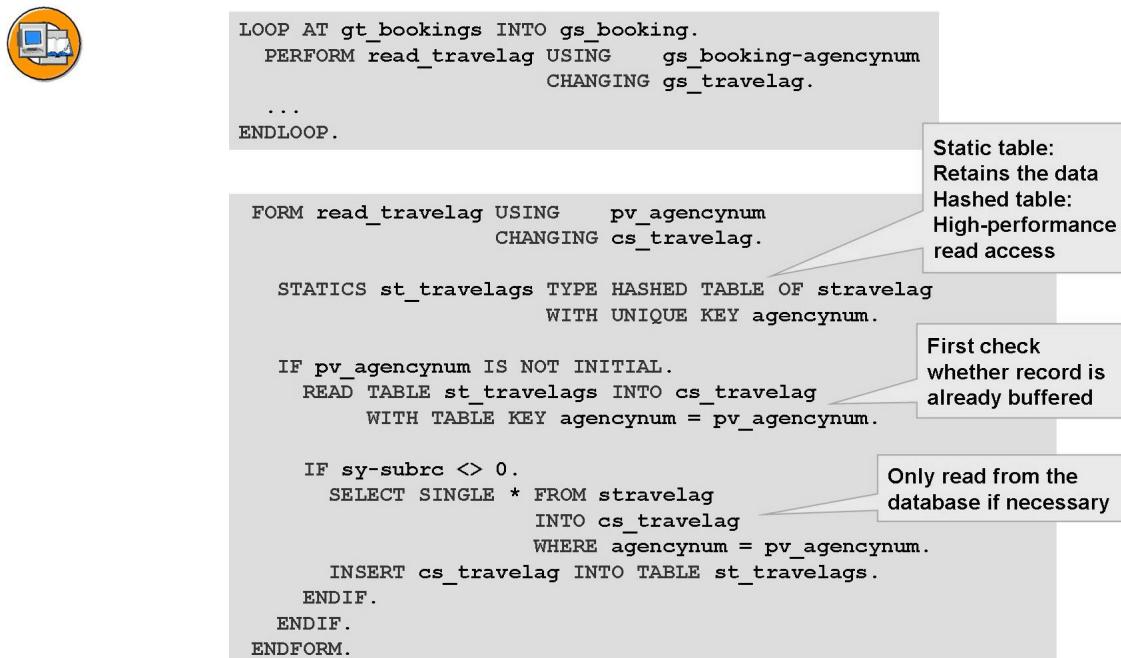


**Hint:** To circumvent the second criterion, you can open two cursors for **the same** database table and have the outer cursor use the *count(\*)* aggregate function to determine the number of records in each group.

## Buffering Techniques

Of course, avoiding database accesses in the first place is the most effective way to reduce database load. You can do this by taking table contents that have already been read and saving them in an internal table (with type SORTED or HASHED, if possible). In many cases, the application logic does not permit the use of joins. If you still need to read data from multiple tables in such cases, you can use a separate internal table to buffer the read data.

### Read on Demand and Buffering



**Figure 208: Read on Demand and Buffering**

In the example, you want to read the matching data from STRAVELAG (travel agency master data) within a loop for the bookings from table SBOOK. You can accomplish this with a `SELECT SINGLE` statement or a read routine. If you use `SELECT SINGLE` in this constellation, you would execute identical SQL statements. You should therefore encapsulate the reading of STRAVELAG records in a read routine. In the example, the table contents read from STRAVELAG are buffered in a static internal table. Before each database access, the system checks whether the corresponding table entry has already been read.

The read routine can also be a method of a local or global class, instead of a subroutine. In this case, a private attribute of the corresponding class is used as buffer instead of the static internal table `st_travelags`.

## FOR ALL ENTRIES Addition

SELECT .. FOR ALL ENTRIES was created in Open SQL at a time when it was not yet possible to perform database JOINS (this was not supported for all SAP-approved DBMS). The connection between the inner and outer database tables is created in ABAP.

Nowadays, this technique is often used when some data is already available in an internal table but additional data is to be read from the database. In such cases, it replaces a SELECT SINGLE statement inside a LOOP over the internal table and normally shows a much better performance than such a loop.

 **Note:** A possible scenario is a program exit in which data can be added to an internal table provided by the SAP standard program.

The following figure gives an example of how the statement works:



### Join between itab and DB Table

```
* fill gt_outer with
* travel agency numbers

SELECT agencynum name
FROM stravelag
INTO TABLE gt_travelags
FOR ALL ENTRIES IN gt_outer
WHERE agencynum =
gt_outer-agencynum.
```

**Caution!**  
If driving table gt\_outer is empty, all entries in DB table STRAVELAG are read!

If gt\_outer is not distinct, values will be read more than once from DB and removed from the final result

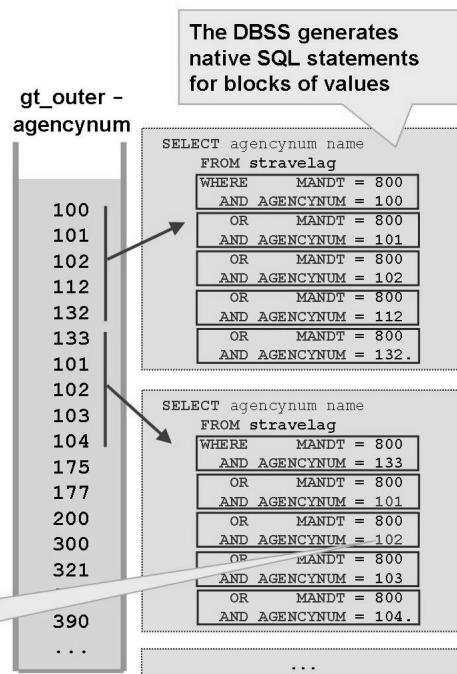


Figure 209: SELECT with FOR ALL ENTRIES Addition

The content of the internal table (driving table) is used as a restriction for the database access. The way this is done may depend on the database system. In general, the database interface takes a certain number of entries in the internal table (in the example it takes 5) and sends one native SQL statement the database for each group. In the end, the results of the individual Native SQL statements are combined to form the result of the Open SQL statement. If the result contains duplicate entries, these duplicate entries are automatically removed. Note, that the size of the packages is controlled by a profile parameter.



**Caution:** If you use FOR ALL ENTRIES, you have to make sure that the driving table is not blank. If the table is blank it provides no boundary condition for the SELECT statement and often the full database table is read. This can cause very long execution times for the statement.



**Hint:** Because duplicates are automatically removed from the result, it is not necessary that the driving table is free of duplicate entries. However, you can improve the performance of the statement if you condense the driving table before the FOR ALL ENTRIES statement (maybe by using a copy of the table). If you do not remove duplicate entries from the driving table, the same data will be read multiply from the database and only be removed in the final combination of the result.

If you want to read **large data volumes, only in exceptional cases** should you use FOR ALL ENTRIES.



# Exercise 19: Implementing a Join for Three Tables

## Exercise Objectives

After completing this exercise, you will be able to:

- Formulate joins over multiple tables
- Explain the difference between inner and outer joins
- Understand why it makes no sense to use joins for buffered tables

## Business Example

Your company uses a custom program that displays a list of bookings for a selection of travel agencies, customers, and flight data.

Unfortunately, your users frequently complain about long runtimes. An analysis shows that the poor performance is due to numerous single record accesses of tables SCARR and STRAVELAG within a loop.

You have been asked to redesign the program so it can read the necessary data in one consolidated database access.

### Template:

BC402\_DBT\_3JOIN

### Solution:

BC402\_DBS\_3JOIN

### Task 1:

Copy the template BC402\_DBT\_3JOIN to the name **ZBC402\_##\_3JOIN**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. From which database tables does the program read the data? Which ones are read in a loop?

---

---

---

3. The database administrators detected the repeated single record accesses for tables SCARR and STRAVELAG, but not for SCUSTOM. Do you have an explanation for this?

---

---

---

4. In this situation, does it make sense to combine the access to SCUSTOM with an access to the other tables in a JOIN?

---

---

---

### Task 2:

Implement a database access that reads from several tables together in a JOIN. Select the three tables for which this makes sense. Note where an outer join is needed to ensure the same number of bookings is read. Only read the columns that are needed. Enhance structure type gty\_s\_bookings accordingly with the additional components.

1. Enhance the database access to table SBOOK. Change the FROM clause to read the necessary data from tables SCARR and STRAVELAG in a triple join. Note that SBOOK also contains postings that were not made by travel agencies, which means the *agencynum* field is initial.

*Continued on next page*

2. Which changes do you have to make to the WHERE condition?

---

---

---

3. In the SELECT clause, add the fields you want to read from database tables SCARR and STRAVELAG. Enhance the line type of the internal table (structure type gty\_s\_bookings) with appropriately typed components. Which other changes, aside from those to the SELECT clause, do you have to make?

### Task 3:

Remove or comment out unnecessary statements and declarations. Activate and test your program.

1. Remove or comment out the SELECT statements for SCARR and STRAVELAG.
2. In the WRITE statement, output the additional fields from internal table gt\_bookings.
3. Remove or comment out the declaration of the unnecessary data objects.
4. Activate and test your program.

## Solution 19: Implementing a Join for Three Tables

### Task 1:

Copy the template BC402\_DBT\_3JOIN to the name **ZBC402\_##\_3JOIN**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. From which database tables does the program read the data? Which ones are read in a loop?

**Answer:** The following tables are accessed:

- SBOOK - Flight bookings
- SCUSTOM - Flight customers
- SCARR - Airlines
- STRAVELAG - Travel agencies

The following tables are accessed in the loop:

- SCUSTOM - Flight customers
- SCARR - Airlines
- STRAVELAG - Travel agencies

3. The database administrators detected the repeated single record accesses for tables SCARR and STRAVELAG, but not for SCUSTOM. Do you have an explanation for this?

**Answer:**

Table SCUSTOM uses **single-record buffering**, which means the data for a specific customer is only read from the database once on a given server.

4. In this situation, does it make sense to combine the access to SCUSTOM with an access to the other tables in a JOIN?

**Answer:** No, because the buffering of SCUSTOM is ignored in a JOIN.

*Continued on next page*

## Task 2:

Implement a database access that reads from several tables together in a JOIN. Select the three tables for which this makes sense. Note where an outer join is needed to ensure the same number of bookings is read. Only read the columns that are needed. Enhance structure type gty\_s\_bookings accordingly with the additional components.

1. Enhance the database access to table SBOOK. Change the FROM clause to read the necessary data from tables SCARR and STRAVELAG in a triple join. Note that SBOOK also contains postings that were not made by travel agencies, which means the *agencynum* field is initial.
  - a) See the source code excerpt from the model solution.

The triple join consists of an inner join between SBOOK and SCARR (field *carrid* is filled for all bookings) and a left outer join between this result and table STRAVELAG.

2. Which changes do you have to make to the WHERE condition?

**Answer:** The *agencynum* field is no longer unique. Therefore, you have to prefix it with the table name, SBOOK, or an appropriate alias.

See the source code excerpt from the model solution.

3. In the SELECT clause, add the fields you want to read from database tables SCARR and STRAVELAG. Enhance the line type of the internal table (structure type gty\_s\_bookings) with appropriately typed components. Which other changes, aside from those to the SELECT clause, do you have to make?

- a) The *carrid* and *agencynum* are no longer unique. Therefore, you have to prefix it with the table name, SBOOK, or an appropriate alias.

See the source code excerpt from the model solution.

## Task 3:

Remove or comment out unnecessary statements and declarations. Activate and test your program.

1. Remove or comment out the SELECT statements for SCARR and STRAVELAG.
  - a) See the source code excerpt from the model solution.
2. In the WRITE statement, output the additional fields from internal table gt\_bookings.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

3. Remove or comment out the declaration of the unnecessary data objects.
  - a) See the source code excerpt from the model solution.
4. Activate and test your program.
  - a) Carry out this step in the usual manner.

## Result

Source code excerpt from the model solution:

```

REPORT  bc402_dbs_3join.

TYPES: BEGIN OF gty_s_booking,
         carrid      TYPE sbook-carrid,
         connid      TYPE sbook-connid,
         fldate      TYPE sbook-fldate,
         bookid      TYPE sbook-bookid,
         customid    TYPE sbook-customid,
         agencynum   TYPE sbook-agencynum,
         carrname   TYPE scarr-carrname,
         agencyname  TYPE stravelag-name,
         agencycity  TYPE stravelag-city,
       END OF gty_s_booking.

TYPES:
  gty_t_bookings TYPE STANDARD TABLE OF gty_s_booking
    WITH NON-UNIQUE KEY
      carrid connid fldate bookid.

DATA:
  gt_bookings  TYPE gty_t_bookings,
  gs_booking   TYPE gty_s_booking.

DATA:
  gv_custname   TYPE scustom-name.
* gv_carrname   TYPE scarr-carrname,
* gv_agencyname  TYPE stravelag-name,
* gv_agencycity  TYPE stravelag-city.

FIELD-SYMBOLS:
  <fs_booking> LIKE LINE OF gt_bookings.

SELECT-OPTIONS :
  so_agy FOR gs_booking-agencynum DEFAULT '100',
  so_cus FOR gs_booking-customid,

```

*Continued on next page*

```

so_fld FOR gs_booking-fldate.

START-OF-SELECTION.

SELECT b~carrid b~connid b~flddate b~bookid
      b~customid b~agencynum
      c~carrname a~name a~city
   FROM sbook AS b INNER JOIN scarr AS c
      ON b~carrid = c~carrid
   LEFT OUTER JOIN stravelag AS a
      ON b~agencynum = a~agencynum
   INTO TABLE gt_bookings
 WHERE b~agencynum IN so_agy AND
       b~customid IN so_cus AND
       b~flddate IN so_fld AND
       cancelled <> 'X'.

LOOP AT gt_bookings ASSIGNING <fs_booking>.

* scustom is buffered - no need for optimizations
   SELECT SINGLE name FROM scustom INTO gv_custname
      WHERE id = <fs_booking>-customid.

*   SELECT SINGLE carrname FROM scarr
*     INTO gv_carrname
*     WHERE carrid = <fs_booking>-carrid.
*
*   SELECT SINGLE name city FROM stravelag
*     INTO (gv_agencyname, gv_agencycity)
*     WHERE agencynum = <fs_booking>-agencynum.

WRITE: /
      <fs_booking>-carrid,
*      gv_carrname,
      <fs_booking>-carrname,
      <fs_booking>-connid,
      <fs_booking>-flddate,
      <fs_booking>-bookid,
      gv_custname,
*      gv_agencyname,
*      gv_agencycity.
      <fs_booking>-agencyname,
      <fs_booking>-agencycity.

ENDLOOP.

```



## Exercise 20: Optional: Buffering Data Completely or On Demand

### Exercise Objectives

After completing this exercise, you will be able to:

- Buffer data completely when it is used in a loop
- Read data in a loop on demand and then buffer it

### Business Example

Your company uses a standard SAP program that creates a list of bookings for a customer. This program has been enhanced as follows to meet your company's needs:

1. An append was used to add several fields to structure type BC402\_S\_BOOKING
2. A BAdI implementation was created to fill the additional fields

Performance problems occur, however, so you have been asked to examine the performance aspects of the BAdI implementation. You quickly discover that single record accesses in a loop are involved. You improve the BAdI implementation by buffering the data in internal tables, to prevent having to read the same records repeatedly.



**Hint:** To avoid making this exercise unnecessarily complicated, we do not use a “real” BAdI. Instead, the BAdI implementation is simulated using a local class. Make sure you only change the definition and implementation of this local class during the exercise. The source code outside of this class represents the SAP program, which you should consider unchangeable.

#### Template:

BC402\_DBT\_BUFFER

#### Solution:

BC402\_DBS\_BUFFER

### Task 1:

Copy the template BC402\_DBT\_BUFFER to the name **ZBC402\_##\_BUFFER**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Which methods does class lcl\_im\_badi have? Which parameters exist in these methods? How are they typed?

---

---

---

3. Display Dictionary structure type BC402\_S\_BOOKING. Which fields were added in an append?

---

---

---

4. Where is method add\_data of class lcl\_im\_badi called?

---

---

---

5. Which database tables are accessed within the local class lcl\_im\_badi?

---

---

---

## Task 2:

Edit the definition of class lcl\_im\_badi. Create two table-type private instance attributes to buffer the data that is read from the database in method add\_data. Create private structure types and table types for this, also in the class definition.

1. Create a structure type in the private section of the class definition whose components include the key and required fields from table (suggested name: **ty\_s\_scarr**).
2. Create a table type, again in the private section of the class definition, that has the structure type as line type (suggested name: **ty\_t\_scarr**).

*Continued on next page*

3. Which table type is particularly suited to this type of buffering?

---

---

---

4. Create an instance attribute in the private section of the class definition (suggested name: **mt\_carriers**) and type it with this table type.
5. Create an appropriate structure type, table type, and instance attribute to buffer the data from database table STRAVELAG (suggested names: **ty\_s\_stravelag**, **ty\_t\_stravelag**, **mt\_travelags**).

### Task 3:

Now edit the implementation of method `add_data`.

Make sure the data is buffered in the instance attributes of the class after it is read from the database.

Full buffering is possible for one of the tables. Which one? Make sure that all entries in this database table are read and buffered the first time the method is called.

Full buffering does not make sense for the other database table. Why not? Make sure the data from this table is read and buffered individually on demand.

1. Which database table is a candidate for full buffering? Why?

---

---

---

2. Program a database access in which the database table that you want to buffer is loaded completely into the corresponding attribute. Ensure that the access is only made during the first call of the `add_data` method.
3. Replace the single record access to the database table with an access to the internal table (the instance attribute).
4. Make sure that each time the database table that you do not want to buffer is accessed (on a single record basis), the individual data is buffered in the instance attribute.

*Continued on next page*

5. Implement an access to the internal table (the instance attribute) before the database access. Make sure the database access is skipped if the corresponding record is already found in the internal table.
6. Activate and test your program.

## Solution 20: Optional: Buffering Data Completely or On Demand

### Task 1:

Copy the template BC402\_DBT\_BUFFER to the name **ZBC402\_##\_BUFFER**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Which methods does class lcl\_im\_badi have? Which parameters exist in these methods? How are they typed?

**Answer:** The class only has one method: add\_data.

The only parameter is changing parameter cs\_booking, which is typed with Dictionary structure type BC402\_S\_BOOKING.

3. Display Dictionary structure type BC402\_S\_BOOKING. Which fields were added in an append?  
**Answer:** CARRNAME, AGENCYNAME, AGENCYCITY.
4. Where is method add\_data of class lcl\_im\_badi called?  
**Answer:** Within a loop of internal table gt\_bookings.

5. Which database tables are accessed within the local class lcl\_im\_badi?

**Answer:** The following tables are accessed:

- SCARR - Airlines
- STRAVELAG - Travel agencies

### Task 2:

Edit the definition of class lcl\_im\_badi. Create two table-type private instance attributes to buffer the data that is read from the database in method add\_data. Create private structure types and table types for this, also in the class definition.

1. Create a structure type in the private section of the class definition whose components include the key and required fields from table (suggested name: **ty\_s\_scarr**).
  - a) See the source code excerpt from the model solution.
2. Create a table type, again in the private section of the class definition, that has the structure type as line type (suggested name: **ty\_t\_scarr**).
  - a) See the source code excerpt from the model solution.

*Continued on next page*

3. Which table type is particularly suited to this type of buffering?
- Answer:** A hashed table, since the same unique key is always used to access the content later in single record access.
4. Create an instance attribute in the private section of the class definition (suggested name: **mt\_carriers**) and type it with this table type.
    - a) See the source code excerpt from the model solution.
  5. Create an appropriate structure type, table type, and instance attribute to buffer the data from database table STRAVELAG (suggested names: **ty\_s\_stravelag**, **ty\_t\_stravelag**, **mt\_travelags**).
    - a) See the source code excerpt from the model solution.

### Task 3:

Now edit the implementation of method add\_data.

Make sure the data is buffered in the instance attributes of the class after it is read from the database.

Full buffering is possible for one of the tables. Which one? Make sure that all entries in this database table are read and buffered the first time the method is called.

Full buffering does not make sense for the other database table. Why not? Make sure the data from this table is read and buffered individually on demand.

1. Which database table is a candidate for full buffering? Why?

**Answer:** Database table SCARR. It only has a few entries, so the probability that many of these entries will actually be needed is high.

In contrast, database table STRAVELAG is fairly large, and we can assume that a given flight customer has only booked flights at a few travel agencies.
2. Program a database access in which the database table that you want to buffer is loaded completely into the corresponding attribute. Ensure that the access is only made during the first call of the add\_data method.
  - a) See the source code excerpt from the model solution.
3. Replace the single record access to the database table with an access to the internal table (the instance attribute).
  - a) See the source code excerpt from the model solution.
4. Make sure that each time the database table that you do not want to buffer is accessed (on a single record basis), the individual data is buffered in the instance attribute.
  - a) See the source code excerpt from the model solution.

*Continued on next page*

5. Implement an access to the internal table (the instance attribute) before the database access. Make sure the database access is skipped if the corresponding record is already found in the internal table.
  - a) See the source code excerpt from the model solution.
6. Activate and test your program.
  - a) Carry out this step in the usual manner.

## Result

Source code excerpt from the model solution:

### **Interface lif\_badi**

```
INTERFACE lif_badi.
METHODS:
  add_data
    CHANGING cs_booking TYPE bc402_s_booking.
ENDINTERFACE.           "lif_badi
```

### **Class lcl\_im\_badi (Definition)**

```
CLASS lcl_im_badi DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_badi.
  PRIVATE SECTION.

    TYPES: BEGIN OF ty_s_scarr,
            carrid TYPE scarr-carrid,
            carrname TYPE scarr-carrname,
          END OF ty_s_scarr.

    TYPES: ty_t_scarr TYPE HASHED TABLE OF ty_s_scarr
          WITH UNIQUE KEY carrid.

    TYPES: BEGIN OF ty_s_stravelag,
            agencynum  TYPE stravelag-agencynum,
            name       TYPE stravelag-name,
            city       TYPE stravelag-city,
          END OF ty_s_stravelag.
```

*Continued on next page*

```

TYPES: ty_t_stravelag TYPE HASHED TABLE OF ty_s_stravelag
      WITH UNIQUE KEY agencynum.

DATA: mt_carriers  TYPE ty_t_scarr,
      mt_travelags TYPE ty_t_stravelag.

ENDCLASS.          "lcl_im_badi DEFINITION

```

## Class lcl\_im\_badi (Implementation)

```

*CLASS lcl_im_badi IMPLEMENTATION.
METHOD lif_badi~add_data.

DATA:
  ls_carrier  LIKE LINE OF mt_carriers,
  ls_travelag LIKE LINE OF mt_travelags.

* carrname - Buffer full
*   SELECT SINGLE carrname FROM scarr
*       INTO cs_booking-carrname
*   WHERE carrid = cs_booking-carrid.

IF mt_carriers IS INITIAL.
  SELECT carrid carrname FROM scarr
    INTO TABLE mt_carriers.
ENDIF.

READ TABLE mt_carriers INTO ls_carrier
  WITH TABLE KEY carrid = cs_booking-carrid.

  cs_booking-carrname = ls_carrier-carrname.

* agencyname and agencycity - buffer on demand

*   SELECT SINGLE name city FROM stravelag
*       INTO (cs_booking-agencyname,
*              cs_booking-agencycity)
*   WHERE agencynum = cs_booking-agencynum.

IF cs_booking-agencynum IS NOT INITIAL.

  READ TABLE mt_travelags INTO ls_travelag

```

*Continued on next page*

```
WITH TABLE KEY agencynum = cs_booking-agencynum.  
IF sy-subrc <> 0.  
  SELECT SINGLE agencynum name city  
    FROM stravelag  
    INTO ls_travelag  
   WHERE agencynum = cs_booking-agencynum.  
  
  INSERT ls_travelag INTO TABLE mt_travelags.  
ENDIF.  
  
cs_booking-agencyname = ls_travelag-name.  
cs_booking-agencycity = ls_travelag-city.  
  
ENDIF.  
  
ENDMETHOD.          "lif_badi~add_data  
  
ENDCLASS.          "lcl_im_badi IMPLEMENTATION
```



## Exercise 21: Optional: Read Additional Data with the FOR ALL ENTRIES Addition

### Exercise Objectives

After completing this exercise, you will be able to:

- Use the FOR ALL ENTRIES addition correctly in the WHERE condition of a SELECT statement

### Business Example

Your company uses a standard SAP program that creates a list of bookings for a customer. This program has been enhanced as follows to meet your company's needs:

1. An append was used to add several fields to structure type BC402\_S\_BOOKING
2. A BAdI implementation was created to fill the additional fields

After a release upgrade, you discover that SAP has enhanced the BAdI with a method that is called before the loop over the bookings.

You realize that you can improve your enhancement by moving the filling of the buffer (instance attributes) to an implementation of this new method.



**Hint:** To avoid making this exercise unnecessarily complicated, we do not use a “real” BAdI. Instead, the BAdI implementation is simulated using a local class. Make sure you only change the definition and implementation of this local class during the exercise. The source code outside of this class represents the SAP program, which you should consider unchangeable.

#### Template:

BC402\_DBT\_FOR\_ALL\_ENTRIES

#### Solution:

BC402\_DBS\_FOR\_ALL\_ENTRIES

#### Task 1:

Copy the template BC402\_DBT\_FOR\_ALL\_ENTRIES to the name **ZBC402\_##\_FOR\_ALL\_ENTRIES**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.

*Continued on next page*

2. Which methods does class lcl\_im\_badi have? Which parameters exist in these methods? How are they typed?

---

---

---

3. Where is the prepare method of class lcl\_im\_badi called?

---

---

---

## Task 2:

Now edit the implementation of method prepare.

Program a database access to each of the database tables SCARR and STRAVELAG. Fill instance attribute mt\_carriers with the data for all airlines and instance attribute mt\_travelags with the data for the travel agencies. Make sure you do not buffer the data for all the travel agencies, but instead only for those you will need later, during the loop pass.

1. Copy the filling of mt\_carriers from method add\_data to method prepare.
2. Is the query as to whether this is the first call still needed?

---

---

---

3. Program a SELECT statement (array fetch) in which you fill instance attribute mt\_travelags. Use the contents of the it\_bookings import parameter to ensure that only the travel agencies that appear in the bookings in it\_bookings are read.

*Continued on next page*

4. Why is it essential to check whether `it_bookings` is filled? What would happen if `it_bookings` were blank?

---

---

---

### Task 3:

Now edit the implementation of method `add_data`.

Remove all the database accesses and replace them through accesses to the instance attributes with the buffered data.

1. Remove the SELECT statement for database table SCARR.
2. Remove the SELECT statement for database buffer STRAVELAG, as well as the filling of the buffer.
3. Activate and test your program.

## Solution 21: Optional: Read Additional Data with the FOR ALL ENTRIES Addition

### Task 1:

Copy the template BC402\_DBT\_FOR\_ALL\_ENTRIES to the name **ZBC402 ## FOR\_ALL\_ENTRIES**, where ## is your group number. Familiarize yourself with the program and how it works.

1. Copy the program and all its subcomponents.
  - a) Carry out this step in the usual manner.
2. Which methods does class lcl\_im\_badi have? Which parameters exist in these methods? How are they typed?

**Answer:** The class has a method called add\_data and another one called prepare.

The only parameter in the add\_data method is changing parameter cs\_booking, which is typed with Dictionary structure type BC402\_S\_BOOKING.

The only parameter in the prepare method is import parameter it\_bookings, which is typed with Dictionary table type BC402\_T\_BOOKINGS.

3. Where is the prepare method of class lcl\_im\_badi called?

**Answer:** Directly before the loop over internal table gt\_bookings.

### Task 2:

Now edit the implementation of method prepare.

Program a database access to each of the database tables SCARR and STRAVELAG. Fill instance attribute mt\_carriers with the data for all airlines and instance attribute mt\_travelags with the data for the travel agencies. Make sure you do not buffer the data for all the travel agencies, but instead only for those you will need later, during the loop pass.

1. Copy the filling of mt\_carriers from method add\_data to method prepare.
  - a) See the source code excerpt from the model solution.
2. Is the query as to whether this is the first call still needed?

**Answer:** No, because the prepare method is only called once, before the loop over the bookings.

*Continued on next page*

3. Program a SELECT statement (array fetch) in which you fill instance attribute mt\_travelags. Use the contents of the it\_bookings import parameter to ensure that only the travel agencies that appear in the bookings in it\_bookings are read.
  - a) See the source code excerpt from the model solution.
4. Why is it essential to check whether it\_bookings is filled? What would happen if it\_bookings were blank?

**Answer:** The FOR ALL ENTRIES addition is only a restriction if the specified internal table is filled. Therefore, if it\_bookings is blank, all records from STRAVELAG are buffered – even though no data at all is needed later, because add\_data is not even executed.

### Task 3:

Now edit the implementation of method add\_data.

Remove all the database accesses and replace them through accesses to the instance attributes with the buffered data.

1. Remove the SELECT statement for database table SCARR.
  - a) See the source code excerpt from the model solution.
2. Remove the SELECT statement for database buffer STRAVELAG, as well as the filling of the buffer.
  - a) See the source code excerpt from the model solution.
3. Activate and test your program.
  - a) Carry out this step in the usual manner.

### Result

Source code excerpt from the model solution:

#### Interface lif\_badi

```
INTERFACE lif_badi.
METHODS:
  prepare
    IMPORTING it_bookings TYPE bc402_t_bookings,
    add_data
    CHANGING cs_booking TYPE bc402_s_booking.
ENDINTERFACE.                                     "lif_badi
```

*Continued on next page*

## Class lcl\_im\_badi (Definition)

```

CLASS lcl_im_badi DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_badi.
  PRIVATE SECTION.
    TYPES: BEGIN OF ty_s_scarr,
      carrid  TYPE scarr-carrid,
      carrname TYPE scarr-carrname,
    END OF ty_s_scarr.

    TYPES: ty_t_scarr TYPE HASHED TABLE OF ty_s_scarr
      WITH UNIQUE KEY carrid.

    TYPES: BEGIN OF ty_s_stravelag,
      agencynum  TYPE stravelag-agencynum,
      name       TYPE stravelag-name,
      city       TYPE stravelag-city,
    END OF ty_s_stravelag.

    TYPES ty_t_stravelag TYPE HASHED TABLE OF ty_s_stravelag
      WITH UNIQUE KEY agencynum.

    DATA: mt_carriers  TYPE ty_t_scarr,
          mt_travelags TYPE ty_t_stravelag.

  ENDCLASS.           "lcl_im_badi DEFINITION

```

## Class lcl\_im\_badi (Implementation)

```

CLASS lcl_im_badi IMPLEMENTATION.
  METHOD lif_badi~prepare.

    * carrname - buffer full
    SELECT carrid carrname FROM scarr
      INTO TABLE mt_carriers.

    * agencyname agencycity - buffer for all entries

    IF it_bookings IS NOT INITIAL.
      SELECT agencynum name city
        FROM stravelag

```

*Continued on next page*

```

        INTO TABLE mt_travelags
        FOR ALL ENTRIES IN it_bookings
        WHERE agencynum = it_bookings-agencynum.

      ENDIF.

ENDMETHOD.                               "lif_badi~prepare

METHOD lif_badi~add_data.

DATA:
      ls_carrier  LIKE LINE OF mt_carriers,
      ls_travelag LIKE LINE OF mt_travelags.

* carrname - read from buffer

*   IF mt_carriers IS INITIAL.
*     SELECT carrid carrname FROM scarr
*       INTO TABLE mt_carriers.
*   ENDIF.

READ TABLE mt_carriers INTO ls_carrier
      WITH TABLE KEY carrid = cs_booking-carrid.

      cs_booking-carrname = ls_carrier-carrname.

* agencyname and agencycity - read from buffer
IF cs_booking-agencynum IS NOT INITIAL.

      READ TABLE mt_travelags INTO ls_travelag
      WITH TABLE KEY agencynum = cs_booking-agencynum.

*   IF sy-subrc <> 0.
*     SELECT SINGLE agencynum name city
*       FROM stravelag
*         INTO ls_travelag
*         WHERE agencynum = cs_booking-agencynum.
*
*     INSERT ls_travelag INTO TABLE mt_travelags.
*   ENDIF.

      cs_booking-agencyname = ls_travelag-name.
      cs_booking-agencycity = ls_travelag-city.

ENDIF.

```

*Continued on next page*

```
ENDMETHOD.          "lif_badi~add_data  
ENDCLASS.          "lcl_im_badi IMPLEMENTATION
```



## Lesson Summary

You should now be able to:

- Use views and formulate joins correctly
- Understand and use subqueries
- Use the FOR ALL ENTRIES addition correctly

# Lesson: Streams and Locators

## Lesson Overview

In this lesson, you learn about the handling of large objects (LOBs) on database tables. Large objects may be large binary strings (BLOBs) or large character strings (CLOBs). The use of data streams and locators for LOBs in database tables can lead to improved performance with regard to the program runtime, by omitting unnecessary data transports, and reducing memory consumption on the application server.



## Lesson Objectives

After completing this lesson, you will be able to:

- Access sub sequences or properties of LOBs on the database table using locators.
- Use locators to copy LOBs on the database table without first copying the data to the application server.
- Process LOBs sequentially by using the stream concept.

## Business Example

You would like to read large strings from a database table and store this information in an internal table or as a file. In addition, you would like to copy large strings on the database. In both cases, you would like to reduce the memory consumption and increase the performance of your program.

## Overview

For normal access to large objects (**LOBs**), ABAP data objects of the types string and xstring are used, into which the entire LOB is transferred for read access, and from which the entire LOB is taken for write access. Depending on the string length, this may lead to a high memory consumption on the application server. For certain operations on the LOB (e.g. copying the LOB on the database table, storing the LOB in a data object different from a string or xstring), it is not necessary to realize the complete LOB in the ABAP program. Memory consumption can be reduced and the program performance with regard to the runtime can be improved by omitting unnecessary data transports. This is permitted by the usage of streaming and / or locator objects (**LOB handles**).



**Hint:** SQL streams and locators are supported as of SAP NetWeaver 7.0 EhP 2.

## Streams

For read access, read streams can be linked to LOBs using the assignment of corresponding streaming objects. The same applies for write access and write streams. LOB data can be partially processed using the methods of the streams. Instances of the following ABAP classes can be linked to LOBs:



- **CL\_ABAP\_DB\_C\_READER:**

An instance of this class may be linked to a large string (CLOB) located on a database table. Used to read CLOB sequentially.

- **CL\_ABAP\_DB\_X\_READER:**

An instance of this class may be linked to a large binary string (BLOB) located on a database table. Used to read BLOB sequentially.

- **CL\_ABAP\_DB\_C\_WRITER:**

An instance of this class may be linked to a large string (CLOB) located on a database table. Used to write CLOB sequentially.

- **CL\_ABAP\_DB\_X\_WRITER:**

An instance of this class may be linked to a large binary string (BLOB) located on a database table. Used to write BLOB sequentially.

To create a streaming class instance, a reference variable for this instance has to be used in the appropriate SQL statement. To read streams, the reference variable has to be used in the INTO clause of the SELECT statement: A LOB from the result set can be assigned to a LOB handle component of a work area, or to an individual reference variable for a LOB handle.

After the execution of the SQL statement the reference variable points to the LOB handle. The related class is determined using the data type of the result set column and the static type of the target variable, or using the CREATING addition if required. The LOB to be read can be evaluated or forwarded using LOB handle methods.



```
* Variant 1: Assigning reader directly to large object (LOB)

DATA lo_reader TYPE REF TO cl_abap_db_x_reader.
DATA lv_field1 TYPE <db table>-field1.
DATA lv_field2 TYPE <db table>-field2.
.

SELECT SINGLE field1 field 2 ... <LOB field>
      FROM <db table>
      INTO (lv_field1, lv_field2, ..., lo_reader)
      WHERE name = ... .
```

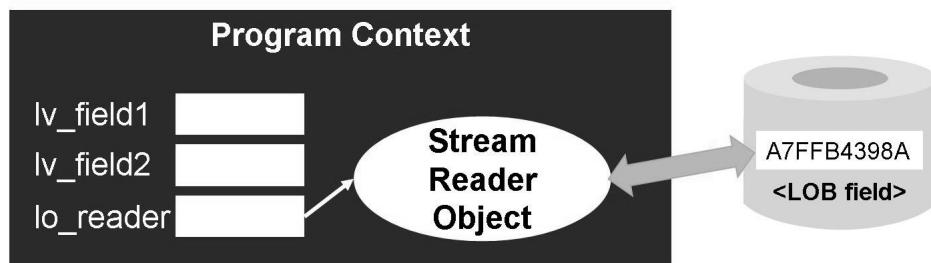


Figure 210: Create Stream Reader Object and assign it to LOB (1)



```
* Variant 2: Result set contains LOB field

DATA ls_db TYPE <db table> READER FOR COLUMNS <LOB field>.

SELECT SINGLE * FROM <db table> INTO ls_db
      WHERE name = ... .

SELECT SINGLE field1 field2 ... <LOB field>
      FROM <db table>
      INTO (ls_db-field1, ls_db-field2, ..., ls_db-<LOB field>)
      WHERE name = ... .
```

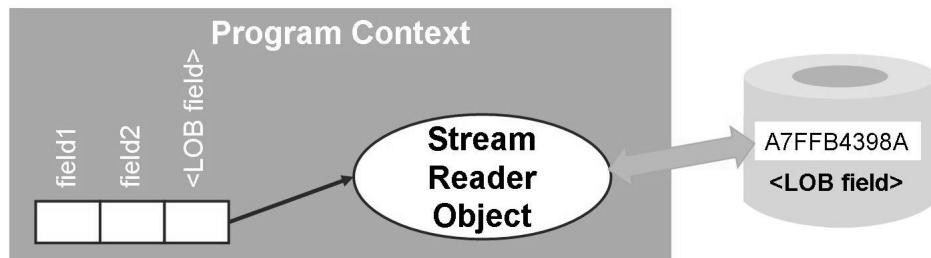


Figure 211: Create Stream Reader Object and assign it to LOB (2)



```
* Variant 3: Stream reader or locator may be assigned to LOB

DATA ls_db TYPE <db table> LOB HANDLE FOR COLUMNS <LOB field>.
DATA lo_handle TYPE REF TO object.

SELECT SINGLE * FROM <db table> INTO ls_db
    CREATING READER FOR COLUMNS <LOB field>
    WHERE name = .....

CREATE OBJECT lo_handle TYPE cl_abap_db_x_reader.
lo_handle ?= ls_db-<LOB field>.

...

SELECT SINGLE * FROM <db table> INTO ls_db
    CREATING LOCATOR FOR COLUMNS <LOB field>
    WHERE name = .....

CREATE OBJECT lo_handle TYPE cl_abap_db_x_locator.
lo_handle ?= ls_db-<LOB field>.
```

**Figure 212: Create LOB Handler and assign it to LOB**

The following important methods are defined in the ABAP classes used to read streams:



- **DATA\_AVAILABLE( )**: Check whether input stream is empty
- **READ( <length> )**: Read the next <length> bytes (characters) from the input stream
- **SKIP( <length> )**: Skip the next <length> bytes (characters)
- **IS\_CLOSED( )**: Check whether data stream is closed
- **CLOSE( )**: Close data stream
- **IS\_MARK\_SUPPORTED( )**: Check whether setting a marker is possible
- **SET\_MARK( )**: Set marker at current reading position
- **RESET\_TO\_MARK( )**: Position read cursor to marking position
- **DELETE\_MARK( )**: Delete marker
- **IS\_RESET\_SUPPORTED( )**: Check whether jumping to the beginning of the LOB is possible
- **RESET( )**: Position reading cursor to beginning of LOB
- **IS\_X\_READER( )**: Check if stream is binary data stream or character data stream
- **GET\_STATEMENT\_HANDLE( )**: Get object describing SQL statement



```

DATA lo_reader  TYPE REF TO  cl_abap_db_x_reader.
DATA lt_picture TYPE TABLE OF string.

SELECT SINGLE picture FROM bc402_blob_table
  INTO lo_reader
  WHERE name = 'BC402_PICTURE'.

WHILE lo_reader->data_available( ) = abap_true.
  APPEND lo_reader->read( 255 ) TO lt_picture.
ENDWHILE.

```

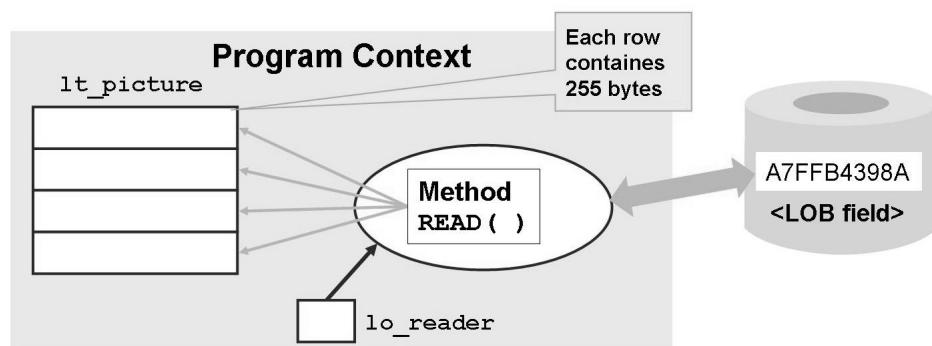


Figure 213: Streams - Example



**Hint:** A maximum of 1000 LOB handles can be open in a database LUW.



**Hint:** Since the number of data streams open at one time is limited to 16, they should be closed as soon as possible using the CLOSE method. Independently of this, **read streams are closed implicitly at the end of a SELECT statement and at the end of a database LUW**. An open write stream can be closed implicitly only by using a database rollback. Database commits for open write streams, on the other hand, cause a non catchable runtime error COMMIT\_STREAM\_ERROR.

## Locators

For read- and write access, locators can be linked to LOBs by assigning corresponding reference variables. Using the methods of the locators, you can access the sub sequences of LOBs or the properties of LOBs without requiring a complete realization in the ABAP program. Furthermore, locators enable the

copying of LOBs within the database without having to transport the data between the database and the application server. Instances of the following classes can be linked to LOBs:



- **CL\_ABAP\_DB\_C\_LOCATOR:**

An instance of this class may be linked to a large string (CLOB) located on a database table. Used to get metadata related to the string, or to read a subsequence of the string.

- **CL\_ABAP\_DB\_X\_LOCATOR:**

An instance of this class may be linked to a large binary string (BLOB) located on a database table. Used to get metadata related to the xstring, or to read a subsequence of the xstring.

To creation of a locator class instance and the assignment of this object to the LOB is similar to the situation for stream reader classes and stream writer classes.

The following important methods are defined in the locator classes:



- **GET\_LENGTH( )**: Returns the LOB's length in characters / bytes
- **FIND( start\_offset = <off> pattern = <pat> )** : Find first occurrence of pattern <pat> in string / xstring. Begin search at offset <off>. Method returns offset of find spot
- **GET\_SUBSTRING( offset = <off> length = <length> )** : Returns substring beginning at offset <off> and having the length <length>
- **IS\_CLOSED( )**: Check whether data stream is closed
- **CLOSE( )**: Close data stream



```

DATA lo_locator TYPE REF TO cl_abap_db_c_locator.
DATA lv_hits      TYPE      i.
DATA lv_length    TYPE      i.

SELECT text FROM sotr_textu INTO lo_locator
  WHERE langu = 'EN'
    AND crea_name = 'EHRETS'.

  IF lo_locator->find( start_offset = 0
                        pattern      = 'error' ) <> -1.
    lv_hits = lv_hits + 1.
    lv_length = lv_length + lo_locator->get_length( ).
  ENDIF.

ENDSELECT.

IF sy-subrc <> 0.
  MESSAGE 'Error processing database table' TYPE 'W'.
ELSE.
  ...
ENDIF.

```

Get all OTR texts in language EN created by user EHRETS

Count occurrences and calculate total length of text containing pattern 'error'

Figure 214: Locator Example - determine Metadata of CLOBs



```

DATA ls_line TYPE bc402_blob_table
  LOCATOR FOR ALL COLUMNS.

SELECT SINGLE picture
  FROM bc402_blob_table
  INTO ls_line-picture
  WHERE name = 'BC402_PICTURE'.

IF sy-subrc <> 0.
  MESSAGE 'Error processing database table' TYPE 'E'.
ENDIF.

ls_line-name = 'BC402_PICTURE' && '_COPY'.
INSERT bc402_blob_table FROM ls_line.

IF sy-subrc = 0.
  MESSAGE 'Copying dataset successful' TYPE 'S'.
ELSE.
  MESSAGE 'Error while copying dataset' TYPE 'E'.
ENDIF.

```

Copy BLOB directly on DB

Figure 215: Locator Example - copy BLOB on Database

## Disadvantages

The use of locators leads to higher resource consumption in the database system. Locators are not yet supported by all databases. In this case, they have to be emulated from the database interface on the application server.

The use of data streams does not lead to increased resource consumption in the database system, but data streams are somewhat more limited in their use. In particular, data streams cannot be used if internal tables are being processed in the Open SQL statements.



## Lesson Summary

You should now be able to:

- Access sub sequences or properties of LOBs on the database table using locators.
- Use locators to copy LOBs on the database table without first copying the data to the application server.
- Process LOBs sequentially by using the stream concept.



## Unit Summary

You should now be able to:

- Name the relevant system components related to SAP Open SQL
- Describe the process flows involved in a database access using Open SQL
- Explain the following terms: database interface, SAP table buffer, database buffer, cursor cache, database index, and optimizer
- Explain the importance of the SAP database interface
- Explain the importance of the possible operators in WHERE conditions
- Use the different operators correctly
- Know the options available for specifying data objects after INTO
- Implement sequential processing of large data volumes
- Request sorted or aggregated data from the database
- Use aggregate functions correctly
- Use views and formulate joins correctly
- Understand and use subqueries
- Use the FOR ALL ENTRIES addition correctly
- Access sub sequences or properties of LOBs on the database table using locators.
- Use locators to copy LOBs on the database table without first copying the data to the application server.
- Process LOBs sequentially by using the stream concept.



I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

# Unit 6

## Analysis and Testing

### Unit Overview

The *ABAP Workbench* provides many different opportunities for testing programs. In this unit, we examine the usage of activatable breakpoints, assertions, and logpoints.



### Unit Objectives

After completing this unit, you will be able to:

- Plan correctness of your ABAP programs
- Illustrate how to improve the maintainability of ABAP coding

### Unit Contents

Lesson: Breakpoints, Assertions, and Logpoints .....	460
Exercise 22: Assertions and Breakpoints .....	469

# Lesson: Breakpoints, Assertions, and Logpoints

## Lesson Overview

In *SAP Web AS 6.20* and later, the (unconditional) breakpoints have been supplemented with conditional assertions, which you learn about in this lesson. You also learn how to group breakpoints and assertions, to activate and deactivate them together. In this context, we also discuss logpoints, which are available in *SAP Web AS 6.40* and later.



## Lesson Objectives

After completing this lesson, you will be able to:

- Plan correctness of your ABAP programs
- Illustrate how to improve the maintainability of ABAP coding

## Business Example

Your company is implementing a major ABAP development project. You are a member of the project team and are responsible for developing a reliable ABAP application. To ensure your ABAP application responds correctly, you use assertions and breakpoints to test it, giving your application the maximum possible stability.

## Overview

Checkpoints define places in a program where the program state can be tested during execution. They are marked with special statements in the source code. Checkpoints can be made activatable by assigning them to a **checkpoint group**.

ABAP supports three types of checkpoints:

### Breakpoints

The **BREAK-POINT** statement defines an unconditional breakpoint. When an active breakpoint is reached, the program flow is interrupted and the debugger starts.

Breakpoints can be made activatable, but can also be created without assignment to a checkpoint group.

## Assertions

The **ASSERT** statement defines a conditional checkpoint. It is always linked with a logical condition. When an active assertion is reached, the system checks the logical condition. If the condition is not true, the assertion is violated and the program is terminated with a runtime error (dump). If assigned to a checkpoint group, you can also select a different response for violated assertions (see below).

Assertions are available in *SAP Web AS 6.20* and later.

## Logpoints

The **LOG-POINT** statement defines an unconditional checkpoint that serves merely to log when a certain point in the source coding is reached. A logpoint can never interrupt or terminate a program. Logpoints must always be assigned to a specific checkpoint group, which means they can always be activated.

Logpoints are available in *SAP Web AS 6.40* and later.

The **ASSERT**, **BREAK-POINT**, and **LOG-POINT** statements for defining checkpoints do not have any effect on the way the program operates; they are used exclusively for test purposes.

## Breakpoints

Apart from being able to execute an ABAP program in the Debugger, you can also start the Debugger call by choosing a breakpoint. This is achieved through placement of one or more watchpoints, called breakpoints, in the program. Breakpoints are places in the program that instruct the ABAP runtime processor to interrupt program execution at a specific place in the source text. The Debugger is not activated until one of these positions is reached. If you want to set a static breakpoint, use ABAP keyword **BREAK-POINT**. Set the breakpoint in the line where you want to interrupt the program.



**BREAK-POINT .**

- Pauses processing, switches to debugger
- No influence on program
- Is ALWAYS passed
- Unconditional

**Figure 216: Breakpoint Statement: Syntax and Benefits**

When you start the program, the ABAP processor interrupts it when it reaches the breakpoint. You can more easily identify the breakpoints if you number these, for example, **BREAKPOINT 1** or **BREAKPOINT 2**. Static breakpoints are always independent of the user without specifying the user name. The

program is always interrupted as soon as the runtime processor reaches the line containing the breakpoint, regardless of which user executes the program. You can also set **user-specific** static breakpoints. To do this, enter the **BREAK <user\_name>** statement.

It is **essential** that you remove static breakpoints from a program before you transport it to a production system. The extended program check is helpful here: It displays an error message if the program in question contains static breakpoints.

In *SAP Web AS 6.10* and later, you can install activatable breakpoints in your programs, which increases maintainability. The ramifications of this are described further below.



**Caution:** If you do not remove static breakpoints from your program, they are transported to your production system. This can affect users in the production system.

## Assertions

You can implement breakpoints and assertions in complex programs, in order to debug and analyze them. We discussed breakpoints in the previous section. In the following, we examine assertions in more detail.

Assertions serve to verify certain assumptions about the program state at a specific place in the program, and ensure that these assumptions are fulfilled. A new ABAP statement, **ASSERT**, was introduced for this purpose.

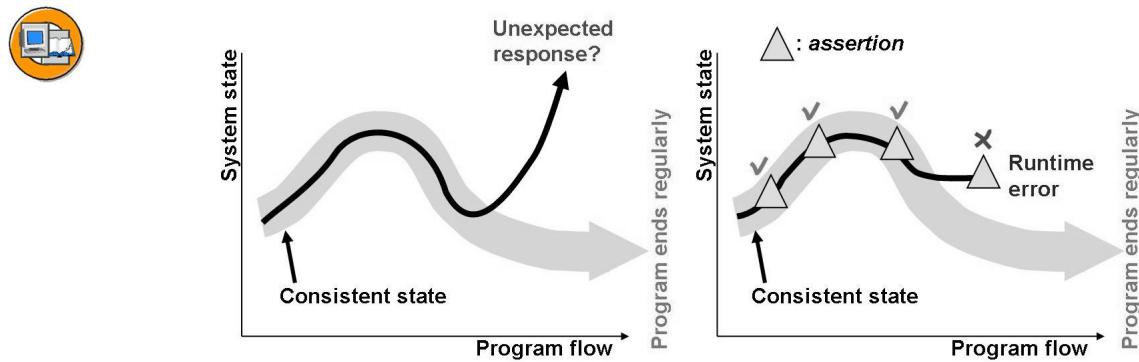


**ASSERT <log\_expr>.**

- **Logical expression is evaluated**
- **If result is true, program is continued**
- **If result is false, a runtime error occurs**

**Figure 217: Assert Statement: Syntax and Program Behavior**

If the logical expression is not fulfilled, the assertion is violated, and a runtime error is raised. In contrast to an IF query, this statement is shorter, its purpose is directly recognizable, and it is activatable (see below).



**Execution is interrupted as soon as an incorrect state is discovered**

**Figure 218: Advantages of Assertions: Early Detection of Deviations from the Expected Program Response**

Roughly speaking, assertions help you check whether a program flow is progressing the way that is intended. The program reacts if its results (or interim results) lie outside of a range of expected values. This you achieve by using assertions. We therefore have another tool for checking the program response, as the example below illustrates.

### Example: Method Call

```
CLASS cl DEFINITION.
...
  CLASS-METHODS find_item IMPORTING item_key      TYPE ty_key
                RETURNING value(item) TYPE ty_item.
...
ENDCLASS.

CLASS cl IMPLEMENTATION.
...
  METHOD find_item.
    READ TABLE item_table INTO item
      WITH KEY table_key = item_key.
  ENDMETHOD.
...
ENDCLASS.

...
my_item = cl->find_item( item_key = my_key ).
```



## Method Call: Verifying Assertion 1

```
...
METHOD find_item.

ASSERT item_key >= 1 AND
    item_key <= LINES( item_table ).

READ TABLE item_table INTO item
    WITH KEY table_key = item_key.

ASSERT item is not initial.
ENDMETHOD.

...
my_item = cl->find_item( item_key = my_key ).
```

Before the method ends, the system checks whether or not an entry was actually found in the internal table.

As you can see, assertions are useful in subroutines: When a subroutine starts, it can check whether certain conditions are met. If not, a runtime error is raised. If the condition is met, processing continues. Another check can be carried out before exiting the subroutine. This enables the subroutine **itself** to check whether it was called properly and whether it returns a plausible result. It can also trigger appropriate reactions itself.

 **Note:** If the subroutine reacts to an incorrect result with an exception, the subsequent program flow depends on whether or not the calling program handles that exception. Therefore, assertions let you form more or less self-contained units.

Exceptions and “assertions” therefore play complementary roles: Exceptions handle **unexpected** conditions, while assertions are suitable for handling **incorrect** application states.

What should you do if a check using assertions is too performance-intensive? In this case, it makes sense to run the assertion in the development and test systems, but skip it in the production system – after all, it has already been tested enough by this point. Activatable assertions and breakpoints are available for this purpose, as discussed in the next section.

## Activatable Checkpoints and Checkpoint Groups

You can design assertions and breakpoints that are not unconditional; instead, the ABAP runtime system only analyzes them under certain circumstances. To do so, you can assign a checkpoint group with the BREAK-POINT and ASSERT statements and the ID addition.



	Always active	Can be activated
<b>Breakpoint</b>	BREAK-POINT. 	BREAK-POINT ID <grp>.
<b>Assertion</b>	ASSERT <log_expr>.	ASSERT ID <group> FIELDS <f1> ... <fn> CONDITION <log_expr>.
<b>Logpoints</b>	not possible	LOG-POINT ID <group> FIELDS <f1> ... <fn>.

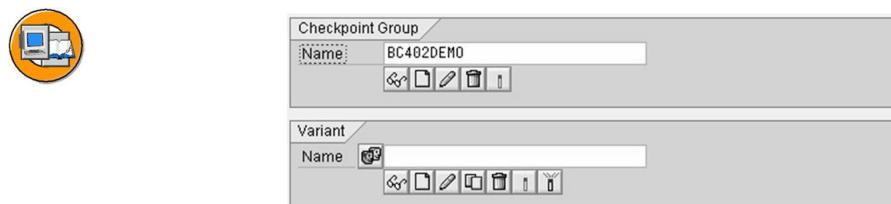
**Figure 219: Classification of Checkpoints**

You use the BREAK-POINT ID <group> or ASSERT ID <group> statement to assign a checkpoint group that contains the information "Active" or "Inactive".

The LOG-POINT statement must always be used with the ID addition.

### Checkpoint Groups

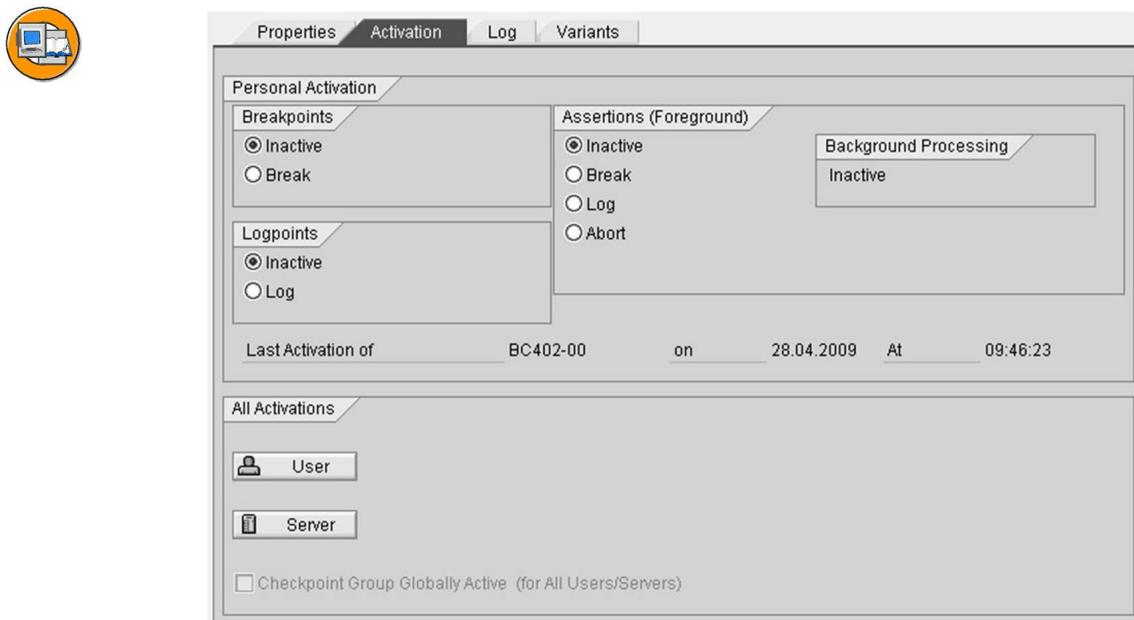
Checkpoint groups represent a new type of Repository object. You can create them in the *Object Navigator* (be sure to observe the naming conventions). Alternatively, you can also create checkpoint groups directly in transaction SAAB. To do so, start transaction code SAAB. The initial screen appears.



**Figure 220: Transaction SAAB Initial Screen for Checkpoint Group Maintenance**

Give the checkpoint group a name and choose *Create* (). Enter a short text and save the checkpoint group. You can then maintain the settings for activating and deactivating checkpoints in the next screen.

 **Note:** A checkpoint group is a repository object. Thus, the name has to be chosen in the customer name space.



**Figure 221: Maintenance Screen of Transaction SAAB**

As you can see, you can activate and deactivate breakpoints and logpoints here. When the system encounters an active breakpoint during program execution, program execution is interrupted and the Debugger is called automatically. When the system encounters an active logpoint, this is recorded in the log of the assigned checkpoint group.

The states of assertions are summarized in the chart below. The individual values have the following meanings:

Inactive	Assertion is ignored/not processed.
Break	The system starts the Debugger if the logical expression is false.
Log	If the expression is false, a log entry is written.
Abort	A runtime error is raised.

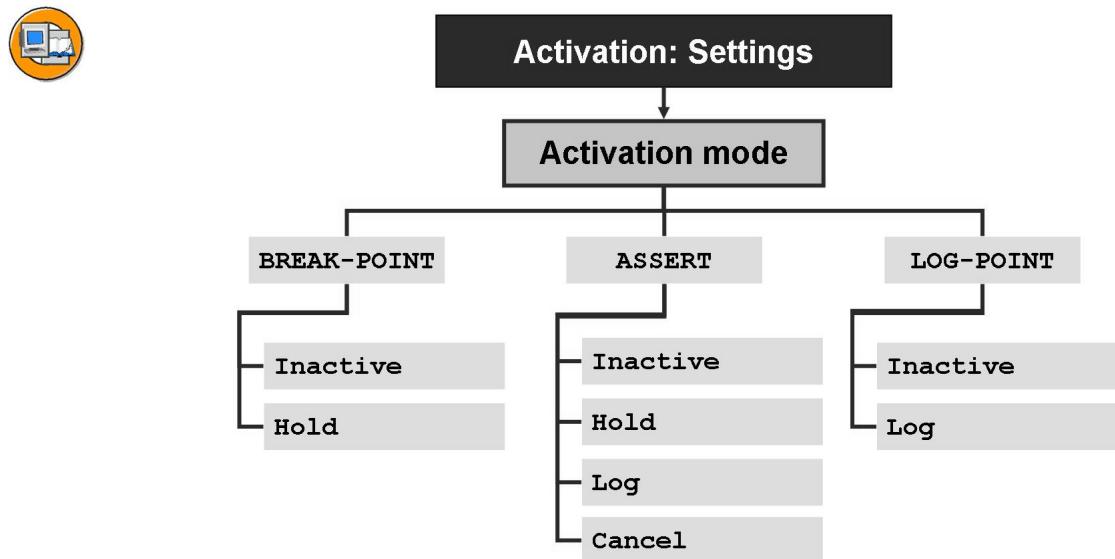


Figure 222: Different Activation Modes for Activatable Checkpoints

For logpoints and logged assertions, the content of any data objects (except for reference variables) can be stored in the log. Here you use the supplement **FIELDS** in the **ASSERT** statement. If *cancel* has been set for assertions, the content of the first eight data objects that are listed behind **FIELDS** are displayed in the generated short dump.



## Exercise 22: Assertions and Breakpoints

### Exercise Objectives

After completing this exercise, you will be able to:

- Define assertions in an application
- Define and analyze activatable assertions and breakpoints

### Business Example

You want to install specific tests in your application, which splits a data string into appropriately typed components.

#### Template:

BC402\_IDS\_DATA\_SET

#### Solution:

BC402\_CPS\_CHECKPOINTS

### Task 1:

Copy program BC402\_IDS\_DATA\_SET (or your own program, ZBC402\_##\_DATA\_SET) to the name **ZBC402\_##\_CHECKPOINTS**, where ## is your group number.

Install assertions in your application to make it more reliable. This will help you detect error states sooner.

1. Copy the program.
2. Add three assertions to your application. Follow the instructions below:

After the function module call, make sure the returned data string is not initial.

After the data string is moved two places to the left, make sure the first character is not "#".

After the identically named fields in the flight structure are filled, make sure the flight date lies in the future.

### Task 2:

Create a checkpoint group (suggested name: **ZBC402\_##**).

1. Create a checkpoint group and enter a descriptive short text.
2. Configure the settings such that

*Continued on next page*

Nothing happens when an activatable breakpoint is reached

Processing is interrupted when an activatable assertion is reached (if the program runs in the background, only a log entry should be written).

3. Adjust all assertions in the above-mentioned program, so that these can be activated using your checkpoint group.
4. Adjust the third assertion in such a way that, when you create a log entry, the flight date is also written in the log.
5. Run the program and check the system reaction.
6. Change the settings of the checkpoint group such that the assertions are now logged. Start the program again and check the activation log.

## Solution 22: Assertions and Breakpoints

### Task 1:

Copy program BC402\_IDS\_DATA\_SET (or your own program, ZBC402\_##\_DATA\_SET) to the name **ZBC402\_##\_CHECKPOINTS**, where ## is your group number.

Install assertions in your application to make it more reliable. This will help you detect error states sooner.

1. Copy the program.
  - a) Proceed as usual.
2. Add three assertions to your application. Follow the instructions below:

After the function module call, make sure the returned data string is not initial.

After the data string is moved two places to the left, make sure the first character is not “#”.

After the identically named fields in the flight structure are filled, make sure the flight date lies in the future.

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

### Task 2:

Create a checkpoint group (suggested name: **ZBC402##**).

1. Create a checkpoint group and enter a descriptive short text.
  - a) Create the checkpoint group in the *Object Navigator*, for example, with *Edit Object*.
  - b) Then choose the *More* tab page and enter a name for the checkpoint group.
  - c) Choose *Create* and enter a meaningful short text in the next screen.
2. Configure the settings such that

Nothing happens when an activatable breakpoint is reached

Processing is interrupted when an activatable assertion is reached (if the program runs in the background, only a log entry should be written).

- a) Configure these settings in the maintenance screen.

*Continued on next page*

3. Adjust all assertions in the above-mentioned program, so that these can be activated using your checkpoint group.
  - a) See the source code excerpt from the model solution.
4. Adjust the third assertion in such a way that, when you create a log entry, the flight date is also written in the log.
  - a) See the source code excerpt from the model solution.
5. Run the program and check the system reaction.
  - a) Run the program.
6. Change the settings of the checkpoint group such that the assertions are now logged. Start the program again and check the activation log.
  - a) Configure the settings accordingly and run the program.
  - b) Navigate back into the checkpoint group, to the *Log* tab index.
  - c) If there are already log entries, open the tree and display the details for an entry. (*Display details* pushbutton or corresponding entry in the context menu for the log entry)

## Result

Source code excerpt from the model solution:

```
REPORT  bc402_cps_checkpoints MESSAGE-ID bc402.
```

TYPES:

```
BEGIN OF ty_s_flight_c,
  mandt      TYPE c LENGTH 3,
  carrid     TYPE c LENGTH 3,
  connid     TYPE n LENGTH 4,
  fldate     TYPE n LENGTH 8,
  price      TYPE c LENGTH 20,
  currency   TYPE c LENGTH 5,
  planetype  TYPE c LENGTH 10,
  seatsmax   TYPE n LENGTH 10,
  seatsocc   TYPE n LENGTH 10,
  paymentsum TYPE c LENGTH 22,
  seatsmax_b TYPE n LENGTH 10,
  seatsocc_b TYPE n LENGTH 10,
  seatsmax_f TYPE n LENGTH 10,
  seatsocc_f TYPE n LENGTH 10,
END OF ty_s_flight_c,
```

```
BEGIN OF ty_s_flight,
```

*Continued on next page*

```

carrid      TYPE sflight-carrid,
connid      TYPE sflight-connid,
fldate      TYPE sflight-fldate,
price       TYPE sflight-price,
currency    TYPE sflight-currency,
planetype   TYPE sflight-planetype,
seatsmax    TYPE sflight-seatsmax,
seatsocc    TYPE sflight-seatsocc,
END OF ty_s_flight.

DATA:
gv_datastring      TYPE string,
gv_set_string      TYPE string,
gv_offset          TYPE i,
gs_flight_c        TYPE ty_s_flight_c,
gs_flight          TYPE ty_s_flight.

START-OF-SELECTION.

* retrieve character string with data
CALL FUNCTION 'BC402_CREATE_SEP_STRING'
*      EXPORTING
*          im_number      = '1'
*          im_table_name = 'SFLIGHT'
*          im_separator   = '#'
*          im_unique      = 'X'
* IMPORTING
*          ex_string      = gv_datastring
* EXCEPTIONS
*          no_data        = 1
*          OTHERS         = 2.
IF sy-subrc <> 0.
MESSAGE a038.
ENDIF.

* make sure the string is filled
*-----*
ASSERT ID      bc402_cps
CONDITION gv_datastring IS NOT INITIAL.

* remove leading and trailing separators
*-----*

```

*Continued on next page*

```

* solution 1 - SHIFT, FIND & Offset access
gv_set_string = gv_datastring.
SHIFT gv_set_string BY 2 PLACES.
FIND '##' IN gv_set_string
    MATCH OFFSET gv_offset.
IF sy-subrc = 0.
    gv_set_string = gv_set_string(gv_offset).
ENDIF.

* solution 2 - REPLACE
* gv_set_string = gv_datastring.
* REPLACE ALL OCCURRENCES OF '##'
*     IN gv_set_string WITH ''.

* solution 3 - SHIFT CIRCULAR & SHIFT
* gv_set_string = gv_datastring.
* SHIFT gv_set_string RIGHT CIRCULAR BY 2 PLACES.
* SHIFT gv_set_string LEFT BY 4 PLACES.
*
* solution 4 - SHIFT DELETING
* gv_set_string = gv_datastring.
* SHIFT gv_set_string LEFT  DELETING LEADING '#'.
* SHIFT gv_set_string RIGHT DELETING TRAILING '#'.
* SHIFT gv_set_string LEFT  DELETING LEADING ' '.

* make sure first character is not '#'
*-----*
ASSERT ID          bc402_cps
CONDITION gv_set_string(1) <> '#'.

* split into (charlike) fragments corresponding to components
SPLIT gv_set_string AT ' #' INTO
    gs_flight_c-mandt
    gs_flight_c-carrid
    gs_flight_c-connid
    gs_flight_c-fldate
    gs_flight_c-price
    gs_flight_c-currency
    gs_flight_c-planetype
    gs_flight_c-seatsmax
    gs_flight_c-seatsocc
    gs_flight_c-paymentsum
    gs_flight_c-seatsmax_b
    gs_flight_c-seatsocc_b
    gs_flight_c-seatsmax_f

```

*Continued on next page*

```
gs_flight_c-seatsocc_f.

* convert fragments into proper data types
MOVE-CORRESPONDING gs_flight_c TO gs_flight.

* make sure flight date not in the past (will not always be the case)
*-----*
ASSERT ID      bc402_cps
FIELDS      gs_flight-fldate
CONDITION  gs_flight-fldate > sy-datum.

* output result
WRITE:
/
gs_flight-carrid,
gs_flight-connid,
gs_flight-fldate DD/MM/YYYY,
gs_flight-price CURRENCY gs_flight-currency,
gs_flight-currency,
gs_flight-planetype,
gs_flight-seatsmax,
gs_flight-seatsocc.
```



## Lesson Summary

You should now be able to:

- Plan correctness of your ABAP programs
- Illustrate how to improve the maintainability of ABAP coding



## Unit Summary

You should now be able to:

- Plan correctness of your ABAP programs
- Illustrate how to improve the maintainability of ABAP coding



I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

I n t e r n a l   U s e   S A P   P a r t n e r   O n l y

# Unit 7

## Hints and Tips – Proven Techniques

### Unit Overview

This unit is a kind of summary of what you have learned so far. Based on the techniques you learned in the preceding units, you get recommendations of how to avoid frequent errors in ABAP programming and what you can do to make your programs readable and maintainable. Lastly, we discuss a few basic rules that apply to the runtime requirements of ABAP programs and database accesses.



### Unit Objectives

After completing this unit, you will be able to:

- Name and avoid frequent mistakes
- Understand and implement the recommendations for robust, maintainable coding
- Explain how excessive runtime requirements can arise during the internal processing of data in programs
- Explain how these runtime requirements can be avoided
- Understand the general performance rules for database access and rate the various access techniques accordingly
- Understand the importance of the WHERE condition for the database optimizer and the access strategy
- Explain, which SELECT statements bypass the table buffer

### Unit Contents

Lesson: Frequent Mistakes and How to Avoid Them .....	480
Lesson: Recommendations and Conventions .....	486
Lesson: Performance (ABAP).....	491
Lesson: Performance (Database Access).....	500

## Lesson: Frequent Mistakes and How to Avoid Them

### Lesson Overview

This lesson describes several frequent mistakes in ABAP programs and shows how you can avoid them proactively.



### Lesson Objectives

After completing this lesson, you will be able to:

- Name and avoid frequent mistakes

### Business Example

You want to develop robust ABAP programs. Therefore, you want to find out how to avoid frequent mistakes in ABAP development.

### Errors Creating and Typing Data Objects



#### Problem:

- **Data objects in dialog modules and event blocks are global**
- **Danger of naming conflicts**
- **When called again, data objects are not initial**

#### Solution:

- **Do not define any data objects in dialog modules and event blocks**
- **Move source text to methods and subroutines and use local data objects there**

Figure 223: Error: Data Objects in Dialog Modules and Event Blocks



#### Problem:

- **TABLES structures are always global**
- **TABLES structures are shared within the program group**

#### Solution:

- **Only use TABLES structures in Dynpro (screen) programming**
- **Never create TABLES structures in processing blocks**
- **Avoid external subroutine calls**

Figure 224: Error: Incorrect Use of TABLES Structures



### Problem:

- Rounding errors due to selection of wrong numeric data type

### Solution:

- Choose numeric data carefully
- When calculating with integer operands, use type p result fields
- Avoid floating point numbers with type f

Figure 225: Error: Wrong Type for Numeric Data Objects

## Errors Using Data Objects

### Character-Type Data Objects



### Problem:

- Illogical values when offset access is made of the entire structure
- Runtime error if offset access is made outside the valid range

### Solution:

- Limit offset access to individual components
- Catch the runtime error (CX\_SY\_RANGE\_OUT\_OF\_BOUNDS)
- Check the offset and length values before access

Figure 226: Error: Offset Access Out of Range



### Problem:

- Program terminations due to invalid values (such as blanks) in data objects with type n, d, and t

### Solution:

- Create data objects with type n, d, or t with valid initial values
- After SPLIT statements, check the data objects for invalid content and initialize them when needed

Figure 227: Error: Invalid Contents with Data Type n, d, t

## Access to References and Field Symbols



### Problem:

- Runtime error when dereferencing references
- Runtime error when accessing field symbols

### Solution:

- Always check validity before dereferencing references
- Always check validity before accessing field symbols

Figure 228: Error: Access to Invalid Pointers

## Access to Internal Tables



### Problem:

- Access to standard table with BINARY SEARCH does not find a line when the table is not sorted

### Solution:

- Always sort the table accordingly before access with BINARY SEARCH

Figure 229: Error: BINARY SEARCH of Unsorted Table



### Problem:

- SELECT statement with addition FOR ALL ENTRIES reads all data when the internal table is empty

### Solution:

- Before accessing the database with FOR ALL ENTRIES, always make sure the internal table is not empty

Figure 230: Error: SELECT with FOR ALL ENTRIES and Empty Internal Table



### Problem:

- Runtime errors due to index access to sorted tables

### Solution:

- Only use key terms for modifying access to sorted tables

Figure 231: Error: Modifying Index Access to Sorted Tables

## Access to System Fields



### Problem:

- Obsolete and internal system fields do not necessarily have logical contents
- Inconsistency after overwriting system fields

### Solution:

- No access to obsolete or internal system fields
- No write access to system fields
- Do not use system fields as actual parameters (particularly during transfer by reference)

Figure 232: Error: Incorrect Access to System Fields



### Problem:

- Inconsistencies
- Security gaps
- Follow-on errors

### Solution:

- Evaluate sy-subrc after every statement that sets the field (possibly with assertion)
- Evaluate sy-subrc immediately

Figure 233: Error: sy-subrc Evaluated Too Late or Not at All

## Access to Interface Parameters



### Problem:

- Export parameters are not necessarily initial during transfer by reference at start of processing block

### Solution:

- Pass on small export parameters as values
- Pass on large export parameters as references, but do not access for reading before the first write access

Figure 234: Error: Read Access to Export Parameters with Transfer by Reference

**Problem:**

- Runtime error in case of incorrect access to
  - Parameters with a generic type
  - Field symbols with a generic type

**Solution:**

- Select the generic type as specifically as possible
- Catch runtime errors (where possible)
- Use RTTI to check the actual type before access

Figure 235: Error: Incorrect Use of Generic Types



## Lesson Summary

You should now be able to:

- Name and avoid frequent mistakes

# Lesson: Recommendations and Conventions

## Lesson Overview

In this lesson, you learn about several recommendations and conventions that help make your programs easier to read - and therefore easier to maintain. You also learn about several general recommendations for designing more robust programs.



## Lesson Objectives

After completing this lesson, you will be able to:

- Understand and implement the recommendations for robust, maintainable coding

## Business Example

You want to develop applications that are easy to read and maintain. Therefore, you want to find out about the recommendations and conventions for developing ABAP programs.

## Conventions for Improved Readability and Ease of Maintenance



### Recommendation:

- **Naming convention for data objects with different visibility (such as "g", "I")**
- **Naming convention for static attributes and instance attributes (such as "g", "m")**
- **Naming convention for types of formal parameters (such as "e", "i", "c", "r", ...)**

### Reason:

- **Improves readability**
- **Avoids conflicts**
- **Avoids overwriting global data objects with local data objects**

Figure 236: Naming Convention for Visibility



### Recommendation:

- Naming convention for the structure of data objects and data types (such as "v", "s", "t" for elementary, structured, table-type)

### Reason:

- Improves readability
- Avoids conflicts
- Avoids confusion

Figure 237: Naming Convention for Data Object Structures



### Recommendation:

- Do not scatter declarations in the source text
- Global declarations at the top of the program
- Local declarations at the top of the processing blocks
- No declarations in dialog modules or event blocks

### Reason:

- Improves readability
- Avoids misunderstandings

Figure 238: Position of Declarations in Source Text



```
TYPES: lty_word TYPE c LENGTH 10.  
DATA: lv_word TYPE c LENGTH 10.
```

Independent type definition

Bound type definition

### Recommendation:

- Define independent data types instead of bound types

### Reason:

- Independent data types have a meaning
- Central maintainability and easier reuse

Figure 239: Independent or Bound Data Types?

**Recommendation:**

- Use constants instead of direct use of literals
- Use text symbols instead of literals (for non-technical texts)

**Reason:**

- Maintainability
- Ease of translation

**Figure 240: Using Constants****Recommendation:**

- Whenever possible, create global data types and constants in global classes and interfaces
- Only create new types in the ABAP Dictionary if Dictionary functions are needed (such as search help for user dialogs)

**Reason:**

- Groups types and constants
- Reference to a context (such as method signature)
- Avoids unintended dependencies between programs

**Figure 241: Global Data Types and Global Constants**

## General Recommendations for Developing Robust Programs

**Recommendation:**

- Use as few global data objects as possible
- Use data as attributes from (local) classes whenever possible
- Make as many attributes as possible private (or at least read-only)

**Reason:**

- Maintainability
- Robustness

**Figure 242: Maximizing Encapsulation of Data Objects**



### Recommendation:

- Do not use external subroutine calls
- Do not use internal tables with header lines

### Reason:

- Obsolete techniques
- Difficult to read
- Error-prone

Figure 243: Obsolete Techniques



```
DATA: lv_chf TYPE c LENGTH 10, " text field
      lv_chs TYPE string,           " text string
      lv_xf  TYPE x LENGTH 10, " byte field
      lv_xs  TYPE xstring.        " byte string
```

### Recommendation:

- Use strings preferentially
- Only use fields when a fixed length is important  
(for example, reference to screen field or database field)

### Reason:

- Strings are more versatile
- Strings only occupy the memory they actually need  
(aside from a small overhead)
- Strings are shared when copied

Figure 244: Fixed Length or Strings?



### Recommendation:

- Only use dynamic programming when absolutely necessary
- Only generate programs if no other techniques are sufficient
- Handle all error cases
- Check prerequisites to avoid runtime errors

### Reason:

- Poor maintainability
- Highly susceptible to errors
- Difficult to test

Figure 245: Using Dynamic Programming Techniques



## Lesson Summary

You should now be able to:

- Understand and implement the recommendations for robust, maintainable coding

# Lesson: Performance (ABAP)

## Lesson Overview

This lesson contains several tips as to how you can minimize the runtime requirements for processing data within ABAP programs. Of course, the correct use of internal tables is particularly important here.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain how excessive runtime requirements can arise during the internal processing of data in programs
- Explain how these runtime requirements can be avoided

## Business Example

You want to develop an application that processes large data volumes and need to minimize the program runtime. You therefore want to find out about the programming techniques that let you avoid excessive runtime requirements proactively.

## General Notes on Performance

Since the internal tables are the largest data objects, they of course play a decisive role in the runtime requirements for internal processing. Nonetheless, there are a few other areas where skilled programming can help improve program performance.

### Avoid Unnecessary Type Conversion

When you define assignments between data objects of different data types, conversion is performed according to specific conversion rules, which generates overhead.



```
DATA lv_int TYPE i.  
DO ... TIMES.  
  lv_int = sy-index.  
ENDDO.
```



```
DATA lv_num TYPE n LENGTH 10.  
DO ... TIMES.  
  lv_num = sy-index.  
ENDDO.
```



```
DATA lv_int TYPE i.  
READ TABLE gt_itab ...  
  INDEX lv_int.
```



```
DATA lv_num TYPE n LENGTH 10.  
READ TABLE gt_itab ...  
  INDEX lv_num.
```



```
CONSTANTS lco_pack TYPE p  
  ... VALUE '0.01'.  
  
DATA lv_pack TYPE p ... .  
DO ... TIMES.  
  lv_pack = lv_pack + lco_pack.  
ENDDO.
```



```
DATA lv_pack TYPE p ... .  
DO ... TIMES.  
  lv_pack = lv_pack + '0.01'.  
ENDDO.
```



**Figure 246: Runtime Losses from Type Conversion**

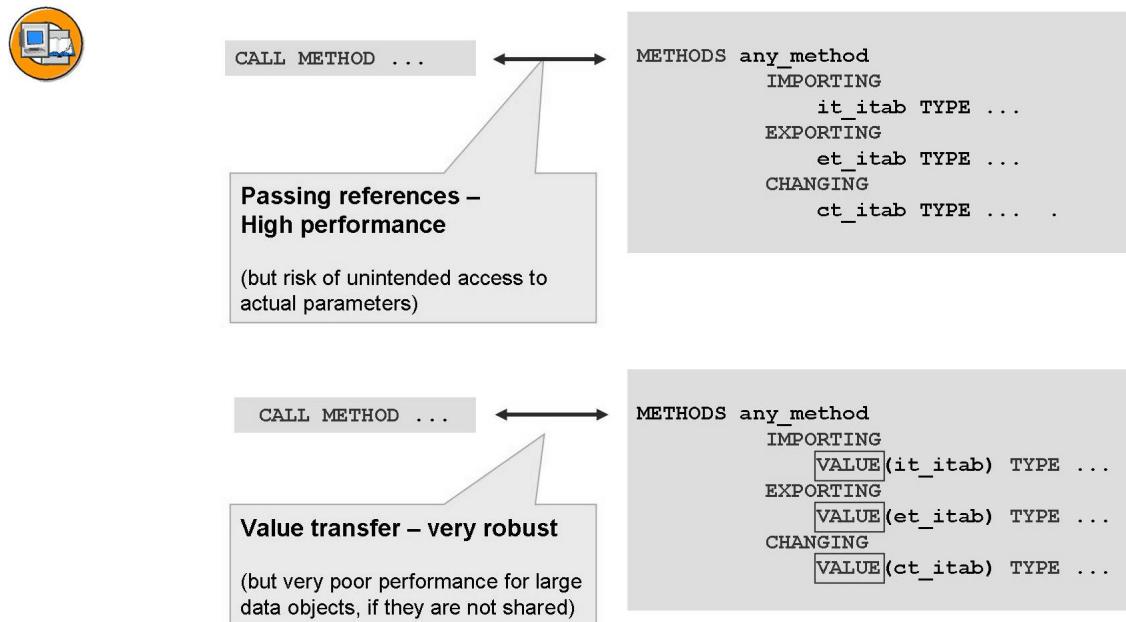
Therefore, you should avoid conversions wherever possible and only assign data objects with the same type to one another.

To avoid unnecessary conversions, always specify the data objects in operand positions that match the operand type.

In cases where type conversions are unavoidable (character literals for non-integer values, for example), create and use a suitably typed constant. In this case, the conversion is only performed once.

### Avoiding Poorly Performing Parameter Transfers

When you define modularization units, you can choose between passing references and passing values for a formal parameter.



**Figure 247: Passing References and Values**

From a performance perspective, passing references is preferred. You have to weigh this against the robustness of passing values on a case by case basis, however.



**Hint:** If sharing is used for passing values in internal tables and strings, this can compensate for the performance disadvantage compared to passing references. However, write access to the formal parameters is not allowed within the processing block in this case, however, as the costs for copying would then be incurred anyway.



## Querying IS SUPPLIED to Avoid Unnecessary Processing

```
FUNCTION bc402_fmdd_divide1_packed.
*-----*
* * Local interface:
* " IMPORTING
* "   VALUE(I_NUMBER1) TYPE DEC8_2 DEFAULT 1
* "   REFERENCE(I_NUMBER2) TYPE DEC8_2
* " EXPORTING
* "   VALUE(E_RESULT) TYPE DEC8_2
* " EXCEPTIONS
* "   ...
* "-----*
```

```
IF i_number1 IS SUPPLIED. "optional parameter
...
ENDIF.
```

```
IF e_result IS SUPPLIED. "export parameter
...
ENDIF.
```

**Figure 248: Example of the IS SUPPLIED Logical Condition in a Function Module**

When you program function modules and methods, you can design the program flow to be dependent on whether the calling program has supplied an optional parameter or not. To do so, you use logical expressions in the format <name> IS SUPPLIED or <name> IS NOT SUPPLIED, where <name> stands for the name of the optional parameter. Note that EXPORT parameters are always optional.

This technique helps you prevent the system from retrieving and formatting data that the caller of the modularization program does not need. This is especially useful when it helps to avoid unnecessary database accesses.

→ **Note:** An obsolete logical condition, IS REQUESTED, still exists for EXPORT and CHANGING parameters.

## Achieving High Performance with Large Internal Tables

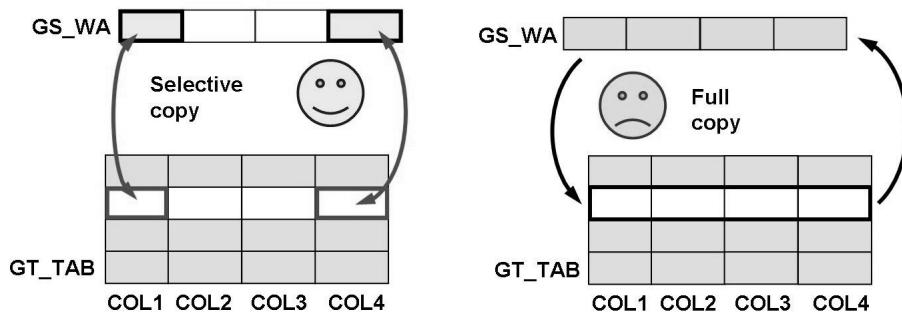
Processing large volumes of data in internal tables can significantly increase the runtime requirements of an ABAP program. Several techniques are available for reducing these runtime requirements.

## Selective Data Transport



```
READ TABLE gt_tab
  INTO gs_wa
  WITH TABLE KEY ...
  TRANSPORTING COL1 COL4.
...
MODIFY TABLE gt_tab
  FROM gs_wa
  TRANSPORTING COL1 COL4.
```

```
READ TABLE gt_tab
  INTO gs_wa
  WITH TABLE KEY ...
...
MODIFY TABLE gt_tab
  FROM gs_wa.
```



**Figure 249: Selective Data Transport**

You can use the TRANSPORTING addition when accessing the contents of internal tables to restrict the copy operations to individual components. This results in a noticeable runtime improvement during repeated accesses. This approach also increases the robustness of modifying accesses, because it helps avoid unintended changes to other fields.

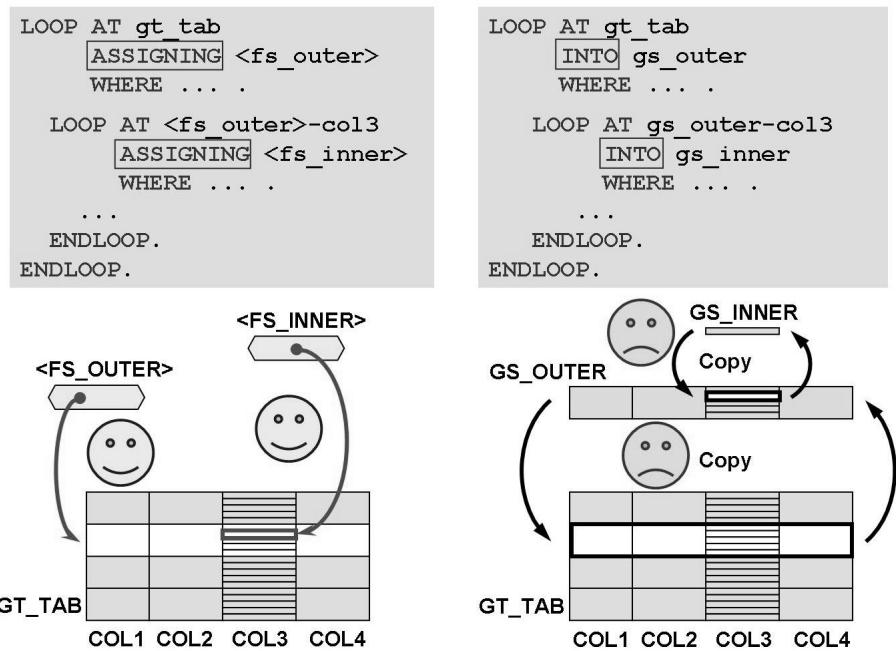


**Hint:** The TRANSPORTING addition is only available in the form TRANSPORTING NO FIELDS for LOOP AT.

## Using Field Symbols and References

When you access internal tables, you can use data references or field symbols as pointers to the table content instead of a work area. This gives you better runtime performance for pure read access with line widths of 1000 bytes or more.

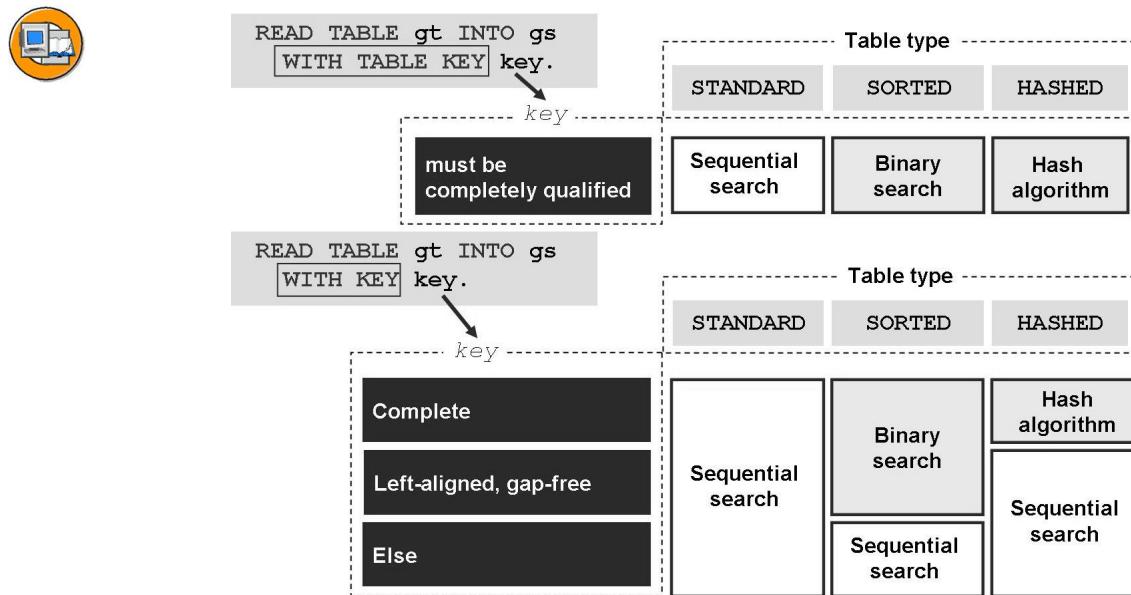
Since the use of field symbols or references gives you direct access to the referenced line, you can make changes to the table contents directly, without the MODIFY statement. As a result, using field symbols or references give better runtime performance for modifying access with line widths of 100 bytes or more.



**Figure 250: Accessing Nested Tables with Field Symbols**

You should always use field symbols or references when nested internal tables are involved. The runtime requirements for a work area (right side of the diagram) are particularly high here, because the entire inner internal table is copied each time (the inner internal tables are not shared). The runtime savings are proportional to the size of the inner internal table.

## High-Performance Single Record Access with Keys



**Figure 251: Fully vs. Partially Qualified Key for Single Record Access**

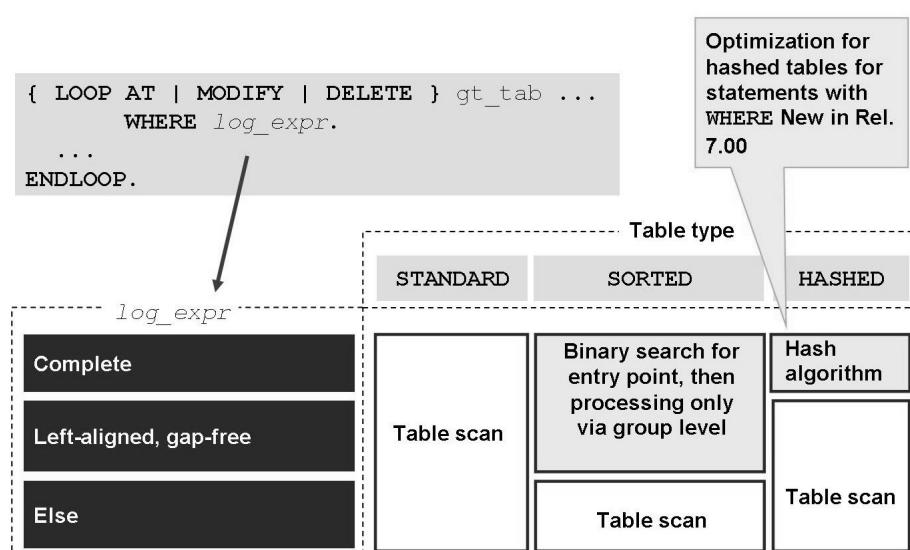
The above diagram summarizes which single record accesses to internal tables can be sped up by using the special sorted and hashed table types. As the diagram shows, the key for hashed tables must be fully qualified to benefit from the hash algorithm.

The ABAP syntax check only verifies compliance with this prerequisite if the key is specified with the **WITH TABLE KEY** addition. To ensure high-performance access for hashed and sorted tables, use this syntax variant whenever possible.

Standard tables are always searched sequentially, regardless of the qualified fields. The **BINARY SEARCH** addition lets you force a binary search. This only returns a correct result, however, if the table was sorted previously by the search fields.



## High-Performance Mass Processing with WHERE



**Figure 252: Mass Processing and Table Types**

As the diagram shows, during mass processing with a WHERE condition, the runtime system compares each table line to see whether it satisfies the specified condition. This is also called a “table scan”. Sorted tables are the only exception to this rule. Runtime optimization is only possible for this table type. Prerequisite: In the WHERE clause, only the first n key fields are filled with a “=” comparison operator (no gaps; n is less than or equal to the number of all key fields).

When looping over a sorted standard table, you can optimize runtime by using single record access with the BINARY SEARCH addition and a loop with a starting point and an explicit termination condition (for more information, see the lesson on “Special Techniques for Using Internal Tables”).



## Lesson Summary

You should now be able to:

- Explain how excessive runtime requirements can arise during the internal processing of data in programs
- Explain how these runtime requirements can be avoided

# Lesson: Performance (Database Access)

## Lesson Overview

This lesson contains several tips as to how you can minimize runtime requirements during database accesses. The correct access to buffered database tables is especially important.



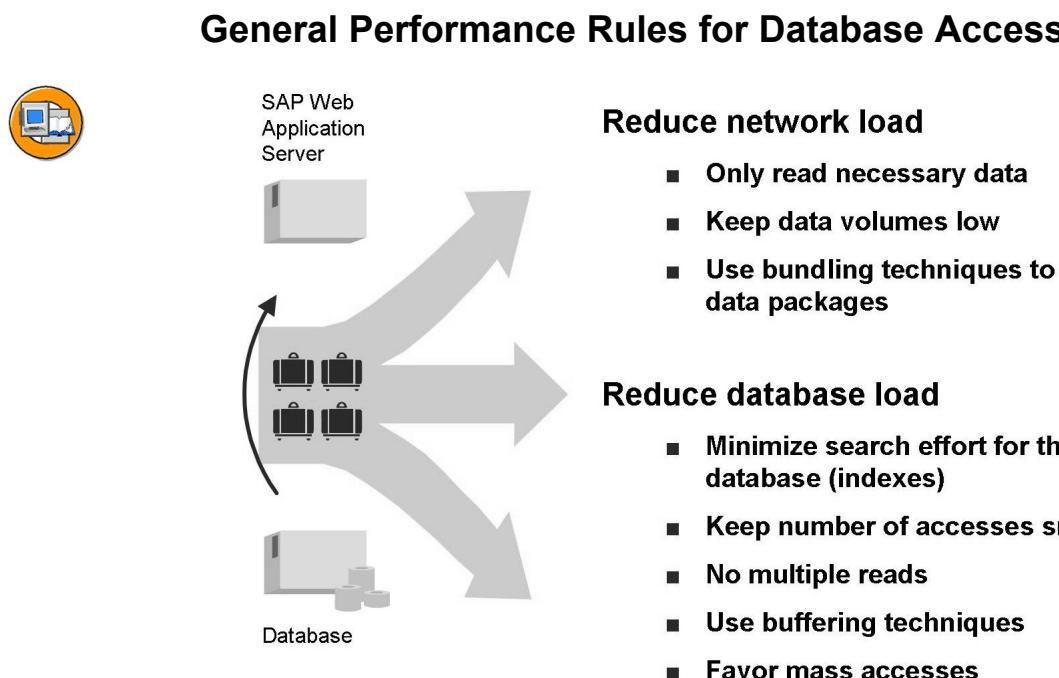
## Lesson Objectives

After completing this lesson, you will be able to:

- Understand the general performance rules for database access and rate the various access techniques accordingly
- Understand the importance of the WHERE condition for the database optimizer and the access strategy
- Explain, which SELECT statements bypass the table buffer

## Business Example

You want to develop an application to process large data volumes. In particular, the program needs to read from the database frequently. You want to minimize the runtime. You therefore want to find out about the techniques for programming database accesses that let you avoid excessive runtime requirements proactively.



**Figure 253: Performance Rules for Database Access**

To optimize performance during database access – for the system as a whole, and not just for this individual program – rule number one is to reduce the load on the network between the application servers and the database, as well as the database server itself.

To do this, the application program must minimize the number of database accesses and keep the amount of data in each transfer small.

To reduce the load on the network between the application servers and the database server, it is important to query data in consolidated packages, and not in individual requests. To do so, you can use buffering techniques, which read all the data that will be needed in the further course of the program. Buffering techniques also help you to avoid multiple reads of the same data.

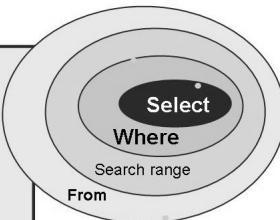
To reduce the load on the database itself, it is important to reduce the search effort required, by using WHERE conditions that support use of the primary or a secondary index.

The diagram below shows how these basic rules can be taken into account when formulating database accesses:



### High-performance applications through:

- **Explicit field list instead of SELECT \***
- **WHERE condition as selective as possible**
- **Support for index use**
  - Index fields in WHERE condition
  - "Good" operators
- **Access to multiple tables instead of nested SELECT statements**
- **Buffer read data in internal tables (avoid accesses with identical structure and values)**
- **Use of SAP table buffering**
- **Changing SQL access as mass access (unless buffered accordingly)**



**Figure 254: Optimizing the Application Logic**

To reduce the data volume, specify a list of fields in the SELECT statement if only a small section of the fields is needed.



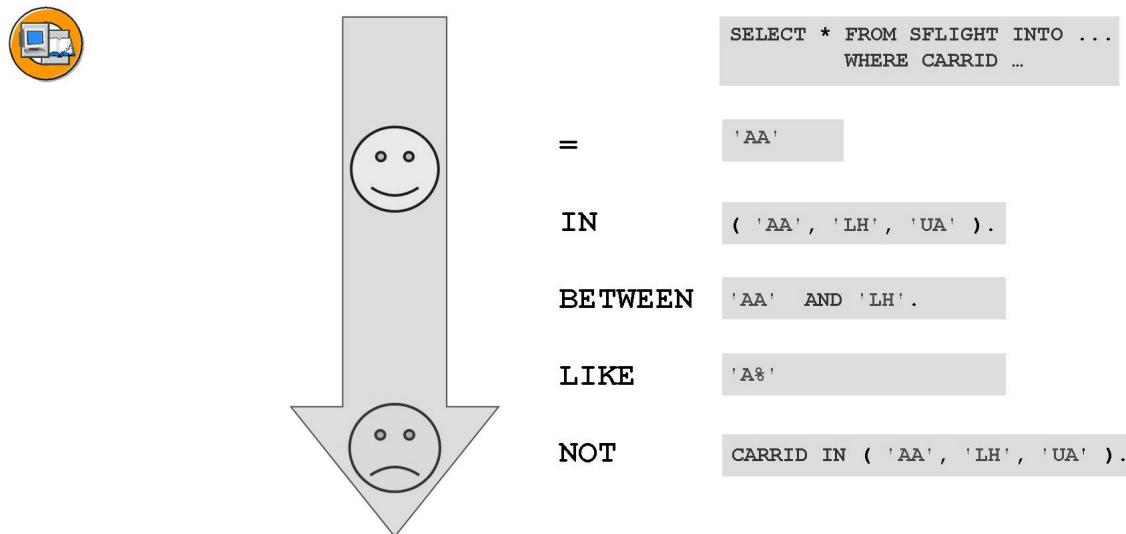
**Hint:** If a large number of fields is needed, performance may be increased by reading the complete lines. However, this limit depends on the respective database.

The WHERE condition should contain as many conditions as possible. You should avoid reading the data first and then deciding which of the data you need. Specific WHERE conditions not only reduce the data volume, but also help the optimizer select a high-performance access strategy.

In this context, we should note that the different operators in the WHERE condition are of varying use for the optimizer.

### Usability of Conditions for the Optimizer

If potential index fields are limited in the WHERE condition, the EQ and = operators are the best for the optimizer. Conditions with negations are the worst. The relative quality of several operators for the optimizer is shown in the diagram below:



**Figure 255: Quality of Operators in the WHERE Condition**

→ **Note:** The diagram is not intended to be a suggestion to avoid certain operators completely; it merely illustrates tendencies in the optimizer results of the various databases that you should be familiar with as a developer.

## Accessing Buffered Database Tables

When you access database tables that use the central SAP table buffering, you have to observe several additional rules.

Firstly, buffering data from these tables within the application does not improve performance.

Secondly, SELECT statements for buffered database tables have to meet certain conditions for the buffering to take effect.

If the SELECT statement contains certain additions or conditions, the SAP table buffer is ignored and the system reads directly from the database instead. Such accesses are said to **bypass** the table buffer.



## Accesses that bypass the buffer:

- **SELECT ... BYPASSING BUFFER**
- **... ORDER BY ...**
- **SELECT DISTINCT ...**
- **Aggregation functions and GROUP BY (HAVING)**
- **Subqueries, ABAP Joins**
- **SELECT FOR UPDATE (set a database block)**
- **Native SQL**
- **WHERE condition with "IS NULL"**
- **For single record-buffered tables:**
  - **Everything but SELECT SINGLE with a fully qualified primary key**
- **For generically buffered tables:**
  - **Everything but SELECT \* ... with WHERE condition in form  
Field = Value for all fields of the generic area**



Figure 256: Statements That Bypass the Table Buffer

In general, these include all Open SQL statements and additions that force an analysis of the database: aggregation functions, sorted reading, join formation, and so on.

You can use the **BYPASSING BUFFER** statement to force a database read explicitly.

In addition, a database access always bypasses the table buffer if there is a possibility that the resulting set is greater than the granularity of the buffering type. If you use a SELECT loop to access a single-record-buffered table, for example, you cannot guarantee that the records you need to read are available **completely** in the buffer. Likewise, the buffer is always bypassed for generically buffered tables if the WHERE condition does not specify unique values for (at least) all the fields in the generic area.



**Caution:** For single-record-buffered tables, a SELECT loop bypasses the buffer even if the WHERE condition is limited to a unique record; use the SELECT SINGLE statement instead for such cases.



## Lesson Summary

You should now be able to:

- Understand the general performance rules for database access and rate the various access techniques accordingly
- Understand the importance of the WHERE condition for the database optimizer and the access strategy
- Explain, which SELECT statements bypass the table buffer



## Unit Summary

You should now be able to:

- Name and avoid frequent mistakes
- Understand and implement the recommendations for robust, maintainable coding
- Explain how excessive runtime requirements can arise during the internal processing of data in programs
- Explain how these runtime requirements can be avoided
- Understand the general performance rules for database access and rate the various access techniques accordingly
- Understand the importance of the WHERE condition for the database optimizer and the access strategy
- Explain, which SELECT statements bypass the table buffer

Internal Use SAP Partner Only



## **Course Summary**

You should now be able to:

- Explain how the ABAP runtime environment works
- Use complex ABAP statements and their variants
- Develop complex ABAP applications with dynamic components
- Use advanced techniques with Open SQL for database access
- Create, test, compare, and classify ABAP applications



# Appendix 1

## ABAP Debugger

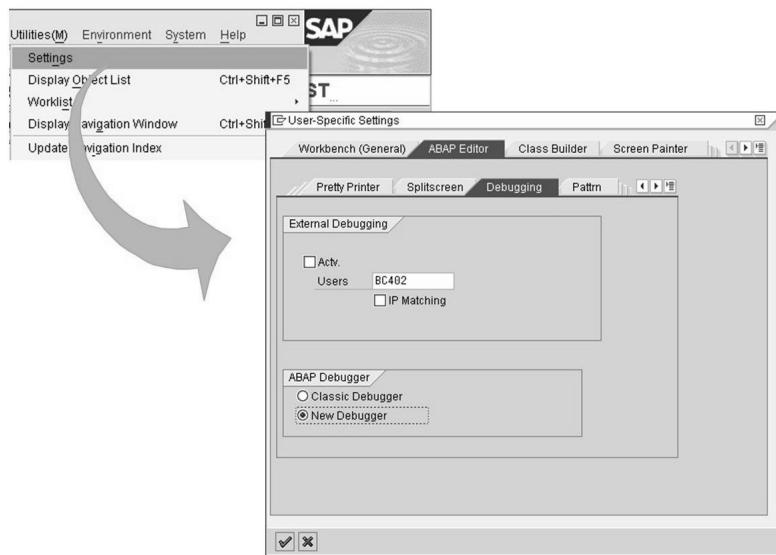
### General Information

A *Debugger* is a programming tool that lets you execute ABAP programs one line or block at a time. It makes it possible for you to display data objects and check the flow logic of programs. There are several options for starting a program in debugging mode from within the *Object Navigator*:

1. Open the context menu in the navigation area of the selected program and choose *Execute → Debugging*.
2. In the edit area, select the program line where you want to start debugging. Choose *Set/Delete Breakpoint*. Then start the program by pressing F8 or opening the context menu in the navigation area and choosing *Execute → Direct*. (Note that you can only set a breakpoint in the editor if the source code is active.)

Two types of debugging are currently possible: Debugging with the conventional Debugger for *SAP NetWeaver 2004* and older releases, or the new Debugger that is available in *SAP NetWeaver 2004* and later.

In the conventional Debugger, the application you are analyzing and the Debugger share the same roll area of an external session, which means every Debugger action can affect the program flow of the application. The new *ABAP Debugger* is executed in a second external (Debugger) session, while the application to analyze (the "debuggee") occupies the original external mode.



**Figure 257: Workbench Settings to Start the New ABAP Debugger**

You can configure the desired Debugger in the *ABAP Workbench* settings. You can switch to the conventional Debugger at any time at runtime.

## The Conventional ABAP Debugger

The conventional *ABAP Debugger* uses the same internal session as the program you are running in debugging mode, which results in certain restrictions. For example, you cannot analyze conversion exits or field exits in the conventional Debugger. Nonetheless, the conventional Debugger has a greater functional scope than the new Debugger at the present time. The most important functions are described below.

### Scanning the Source Code

You can use the following functions to scan the source code using differently sized steps:



### Scanning the Source Code in the ABAP Debugger

Function	Effect
Single Step (F5)	Executes the next statement.
Execute (F6)	Executes the next statement. If the next statement is a subroutine, function module, or method, this routine is executed in total.
Return (F7)	Executes the remainder of a subroutine, function module, or method and returns to the calling position.
Continue (F8)	Executes all statements up to the next breakpoint. If no further breakpoint has been set, the <i>Debugger</i> ends.

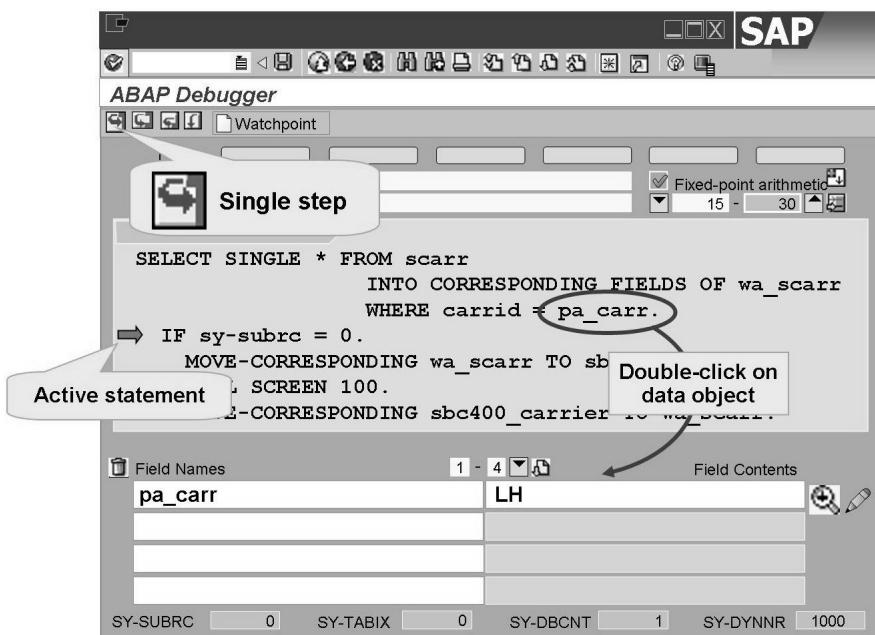


Figure 258: Displaying and Changing Field Contents

### Changing Field Contents

You can double-click a field name to display the contents of that field. When you double-click the field display, the displayed detail view differs depending on the field type. To change field content, edit the value field and press the *Change* button.



**Hint:** To change field content, you need debugging change authorization for authorization object **S\_DEVELOP**.

You can display up to eight fields at the same time in the conventional Debugger. Only the fields of the active program are displayed by default. If you want to display a field from a previously analyzed program, set the program name in parentheses in front of the field name: (program\_name)field\_name; the field contents are displayed.

## Breakpoint Variants

The *ABAP Debugger* differentiates between the following breakpoint variants:



### Breakpoint Variants

<b>Static</b>	Keyword BREAK-POINT inserts a user-independent breakpoint as an ABAP statement in the source code. The statement BREAK <user_name> sets a user-dependent breakpoint in the source code.
<b>Directly set dynamic breakpoints</b>	Are set in the <i>ABAP Editor</i> or the <i>Debugger</i> by double-clicking on a line. Dynamic breakpoints are always user-specific and are deleted when you log off from the SAP system.
<b>Breakpoints for statements</b>	The <i>Debugger</i> interrupts the program directly before the specified statement is executed.
<b>Breakpoints for subroutines</b>	The <i>Debugger</i> interrupts the program directly before a subroutine is called.
<b>Breakpoints for function modules</b>	The <i>Debugger</i> interrupts the program directly before a function module is called.
<b>Breakpoints for methods</b>	The <i>Debugger</i> interrupts the program directly before a method is called.
<b>Breakpoints for exceptions and system exceptions</b>	The <i>Debugger</i> interrupts the program immediately after a runtime error is raised.

## Dynamic Breakpoints

Dynamic breakpoints are breakpoints that you can set without changing the program source code. You can set up to 30 dynamic breakpoints.

In the *ABAP Editor*, choose *Utilities* → *Breakpoint*. The method used to highlight the breakpoint the source code depends on which release you use. You can choose *Utilities* → *Breakpoints* → *Display* to display a list of all dynamic breakpoints in a program. You can use this list to navigate to a specific breakpoint or delete one or all breakpoints from your program source code.

To set a dynamic breakpoint in the **Debugger**, double-click the desired line. In the conventional Debugger mode, double-clicking again deletes the breakpoint you just created. In contrast, in the new Debugger, double-clicking again deactivates

the breakpoint; if you want to delete it, you have to double-click the line once more. When you exit debugging mode, any breakpoints that you have not saved beforehand are deleted automatically.

### Breakpoint for Statement

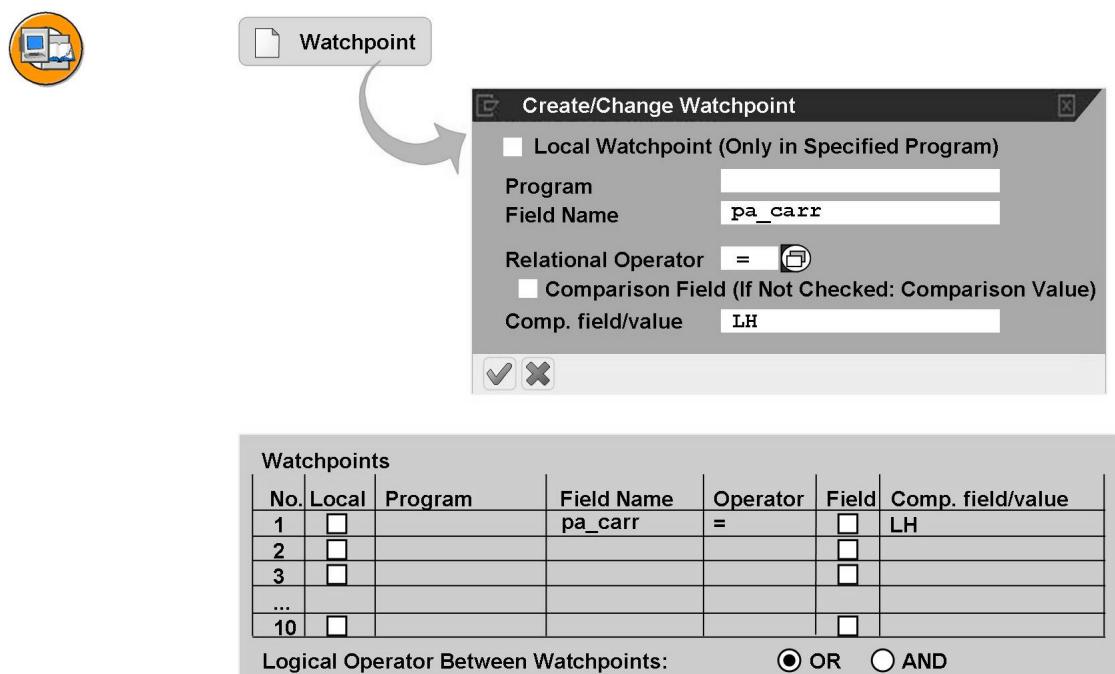
You can set breakpoints at any ABAP statements in the Debugger. To do so, select the corresponding menu item. This makes it possible, for example, to set a breakpoint at every Select statement.

### Breakpoint for Modularization Unit

You can set breakpoints in the *Debugger* that interrupt program execution as soon as a specific modularization unit is reached, such as:

- Subroutines
- Function modules
- Methods

## Watchpoints



**Figure 259: Creating a Watchpoint**

Like breakpoints, watchpoints are places in the program that instruct the ABAP runtime processor to interrupt program execution at a specific place in the source code. In contrast to breakpoints, watchpoints do not activate debugging mode until the contents of a field change. You can only define watchpoints in the *Debugger*.

If you set a watchpoint for a field **without** specifying a relational operator or comparison value and then press *Continue*, program execution continues until **the contents of the field change**.

In contrast, if you specify a **relational operator and comparison value** when you define a watchpoint and then press *Continue*, program execution continues until the **specified condition is met**.

You can set up to ten watchpoints and link them together with a logical operator (AND or OR). You can use watchpoints to display changes to references of strings, data, or objects, as well as internal tables.

If you enter an ampersand (&) before the object name, the respective reference is displayed in hexadecimal notation. If you enter an asterisk (\*) before an internal table name, the table header is also output.

&objectname	Displays the reference of strings, internal tables, or data/object references
*itab	Displays the table header of internal table itab

## Memory Use

You can select menu path *Goto* → *Status Display* → *Memory Use* to display the current memory use. In addition to displaying the memory use directly in the *Debugger*, the memory analysis in this function lets you save memory extracts and analyze them with the *Memory Inspector* (transaction code S\_MEMORY\_INSPECTOR; see to the *SAP Library* for more information).

The following memory usage displays are available:

Memory Use – Total	Detailed display of memory use
Memory Use – Ranked List	List of data objects used, sorted by bound memory
Memory Use – SCC (strongly connected components)	Dynamic objects of an internal session (internal tables, strings, objects)

For more information, see the individual units and the *SAP Library*.

## Sessions

Menu path *Debugging* → *Sessions* makes it possible to permanently save breakpoints and the settings for *System Debugging* and *Always Create Exception Object* and load them again for later reuse. You can assign a ten-place name and expiration date to each session. The expiration date can lie up to one month in the future and can be extended. When you save a session, the session and its

configured settings are available to all users and sessions. If you want to run a source code through the *ABAP Debugger* several times with the same settings, this function reduces the effort needed to set the breakpoints and settings to a minimum.

If a session is no longer required, simply click *Delete* to delete it.

## The New Debugger

The new *ABAP Debugger* is available in *SAP NetWeaver 2004* and later. It replaces the conventional Debugger as of *SAP NetWeaver 7.0*.

### Reasons for the New Debugger

The conventional Debugger runs in the same roll area of the same external session as the application you are analyzing, whereas the new Debugger runs in a separate external session.



### Debugger Comparison

	Conventional Debugger		New Debugger
+	Shared session = Low resource requirements	+	Two sessions = Less interdependency
—	Fixed user interface design	+	Freely configurable user interface
—	Debugging of conversion exits/field exits not possible	+	Debugging of conversion exits/field exits is possible
—	Settings are lost when you exit the Debugger	+	The Debugger exists as long as the external session. Settings are retained.

### Starting and Exiting

You start the new Debugger just like the conventional Debugger:

- With the program context menu in the *Object Navigator*
- By entering /H in the OK code field
- With menu path *System → Services → Debugging ABAP/Screen* (same as /H)

Starting the new Debugger opens a new external session.



**Caution:** If you have more than five active external sessions, you cannot start the new Debugger.



**Figure 260: Initial Screen of the New ABAP Debugger**

As mentioned above, the conventional Debugger exits when you exit the program. The new Debugger is different. We differentiate between the following situations:

- The Debugger is ready for input and the application is waiting for an entry.  
In this case, you have to choose menu path *Debugger → Exit Debugger* to close the Debugger and continue running the application. If you want to exit both the Debugger and the application, choose menu path *Debugger → Exit Application and Debugger*.
- The application is ready for input and the Debugger is waiting for a user action. In this case, enter function code **/hx** to close the Debugger attached to this session.

You can switch from the new Debugger to the conventional Debugger at any time. If you are currently debugging a conversion exit or field exit, however, the system raises runtime error RPERF\_ILLEGAL\_STATEMENT.

## User Interface

You can freely configure the user interface of the new Debugger. Up to nine desktops are available.

## Functions

The new Debugger features the same core functions as the conventional Debugger. You use the same functions to navigate through the source code, for example. In contrast to the conventional Debugger, you can create several breakpoints at the same time in the new Debugger. To do so, choose *Breakpoint → Create Breakpoint*.

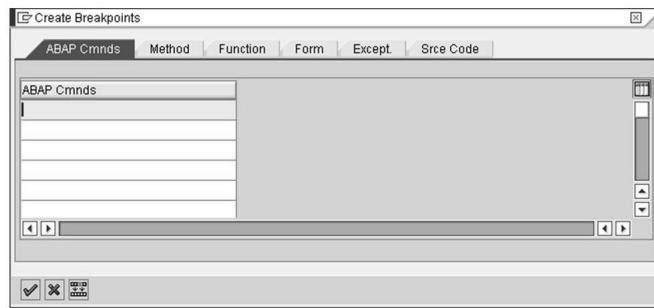


Figure 261: Creating Breakpoints in the New Debugger

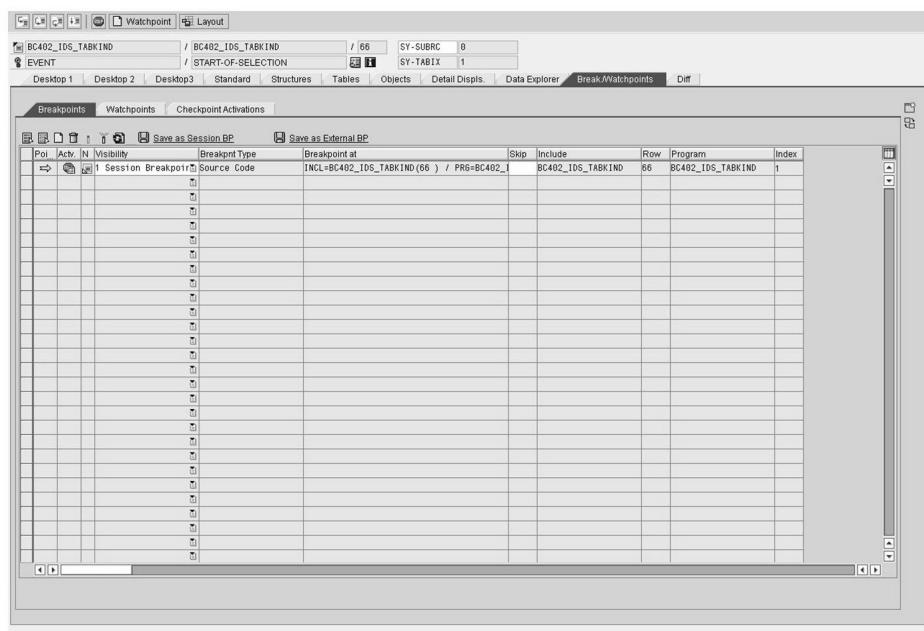
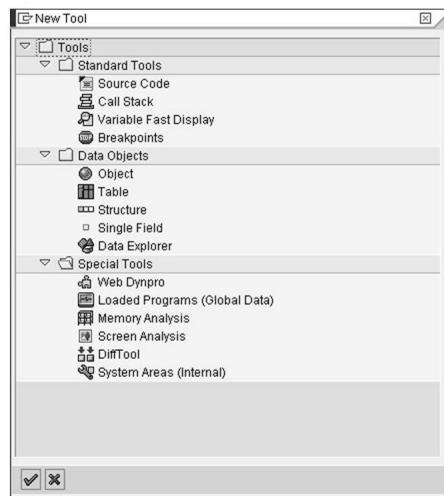


Figure 262: List of Breakpoints in the New Debugger



**Figure 263: Switching Tools**

In addition to setting breakpoints and watchpoints, you can also change the state of activatable checkpoints (see section “Assertions, Breakpoints, and Logpoints”).

# Appendix 2

## Runtime Analysis

### Using the Runtime Analysis

The *runtime analysis* tool makes it possible to take detailed measurements of the runtime requirements of your programs. This lets you locate the source code blocks in your program that are responsible for poor performance and “tune” them accordingly.



**Hint:** Note, however, that measurement results also depend on the current system and network load, as well as the active table buffer and dataset. Since the runtime analysis is usually carried out in development and test systems, you have to qualify the results accordingly.

The diagram below shows how to reach the measurement environment:

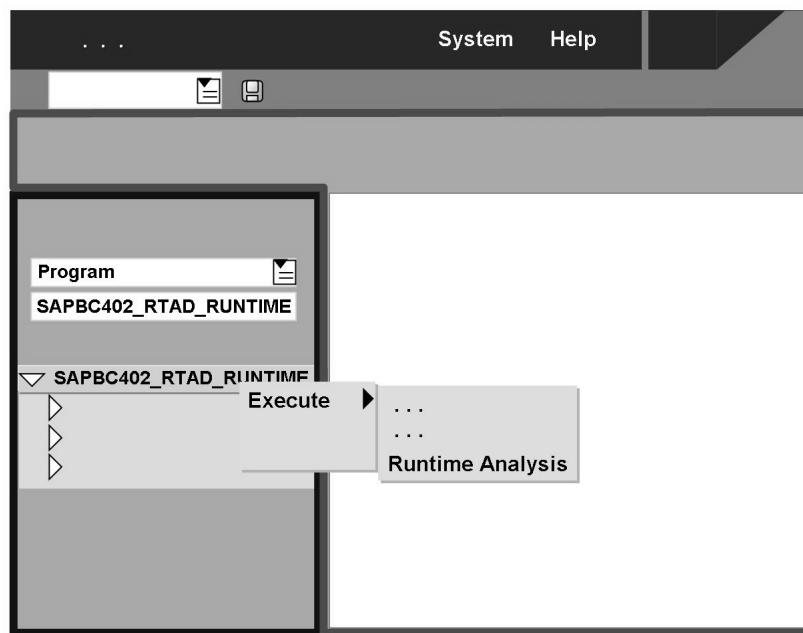


Figure 264: Navigating to the Measurement Environment

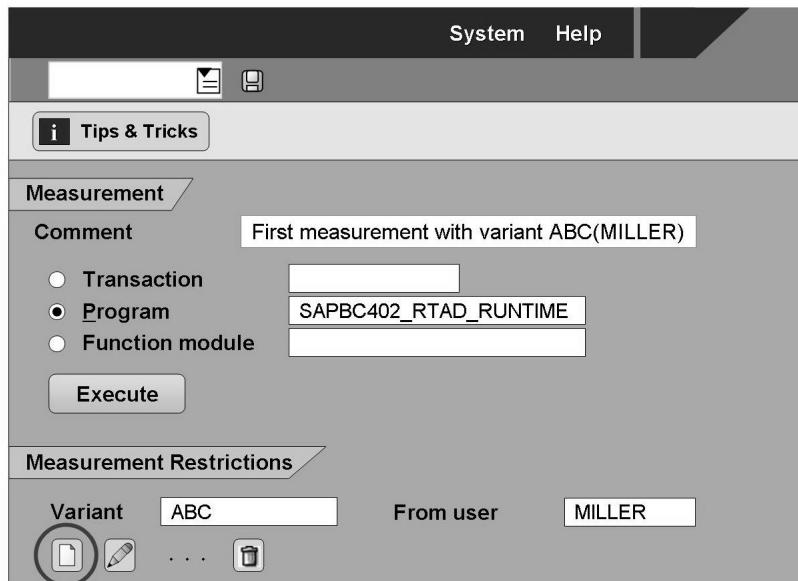


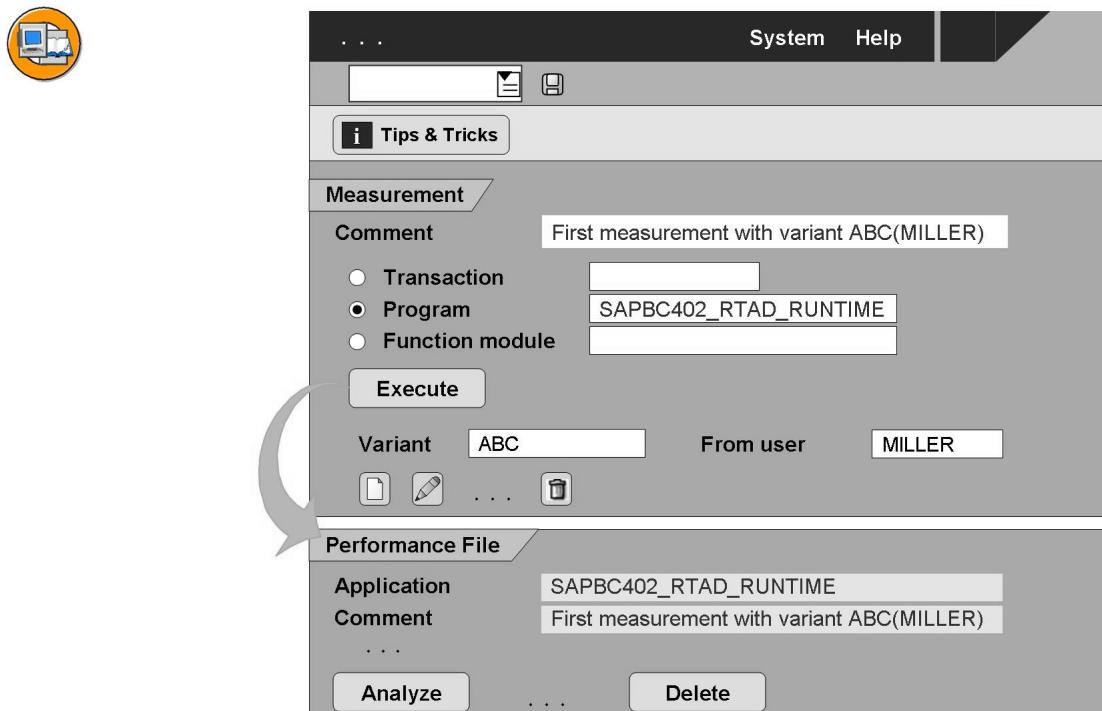
Figure 265: Measurement Environment: Defining a Variant for the Runtime Analysis

In the **initial screen of the runtime analysis (measurement environment)**, you should first enter a short text for your performance measurement. You can then select certain measurements later and compare them with one another.

In addition, you can define detailed measurement criteria as a **variant** (measurement variant) and carry out the measurement as stipulated in your variant.

If you do not specify an explicit variant for a measurement, the system uses the standard variant, named DEFAULT. It is predefined as the default value in the measurement environment; you can click the corresponding button (eyeglasses icon) to display it.

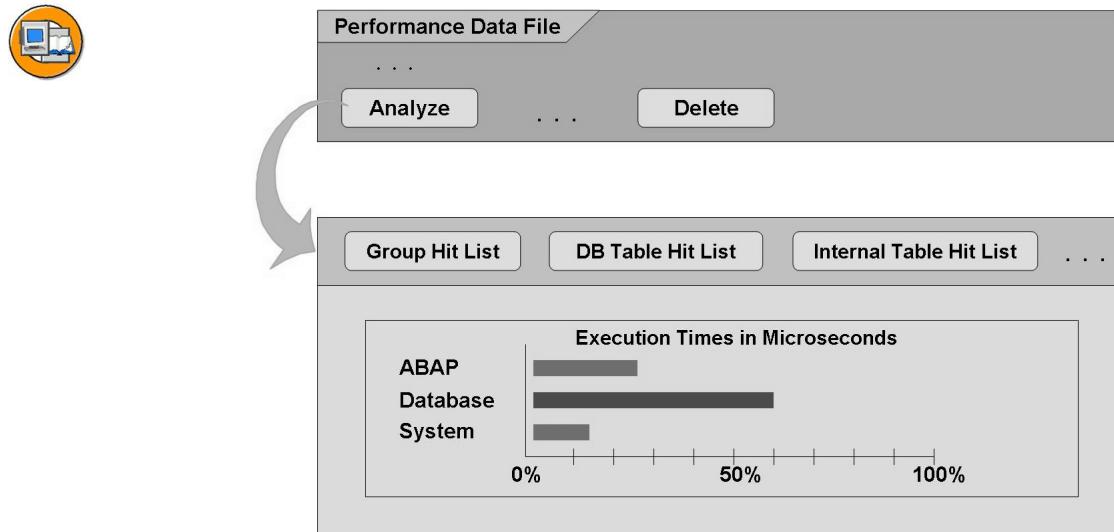
However, we recommend creating your own variant with attribute “*Aggregation = None*” and using it to carry out your measurements, as it provides detailed grouping of the measurement results. You can also specify individual measurement criteria in your variant that restrict the measurement to the relevant program actions, and avoid unnecessary measurement results.



**Figure 266: Measurement Environment: Conducting a Runtime Analysis**

In addition to programs, the *runtime analysis* tool also lets you measure the runtime of transactions and function modules. Select the desired object type and enter the name of the measurement object. In addition, specify the name of the measurement variant to use (if any).

Press the *Execute* button to start the runtime analysis. The generated measurement results are saved in a measurement-specific file, which you can analyze from within the measurement environment.



**Figure 267: Measurement Environment: Evaluating a Runtime Analysis**

After you have completed the runtime analysis, press the *Analyze* button to display the measurement results. The runtime activities are initially divided into three areas:

#### ABAP

All ABAP statements (except database accesses)

#### Database

All database accesses

#### System

Program load processes (including generation, if necessary) and other system activities

If you used a variant with attribute “Aggregation = None” for the measurement, the hit list functions shown in the above diagram is available for even more detailed analysis.

The **Group Hit List** button lets you display the measurement results divided by the following categories: subroutine calls, function module calls, module calls, and method calls, along with operations on internal tables and database tables. To display a detailed breakdown of the information, double-click a measurement result.

The **Internal Table Hit List** and **DB Table Hit List** buttons display measurements of the corresponding table operations. Again, you can double-click individual items to display detailed information.

Certain measurements in the results display are specified in **gross** and **net** figures. The gross figure is the total runtime for the respective action, while the net figure only contains the share of the gross time that is not identified separately as a sub-action runtime.

Special pushbuttons are provided for sorting the measurement results (with regard to runtime requirements) in ascending or descending order. As a result, you can display the most processing-intensive activities in your program at the start of the results list.

To display the ABAP statement for an activity, position the cursor on that activity and press the *Display Source Code* button.



**Hint:** In the initial screen of the runtime analysis, you can press the *Tips & Tricks* button to display a demonstration environment that contains useful performance tips and illustrates the benefits of different source codes by comparing their runtimes.

These tips can help you replace the statements that the runtime analysis has identified as performance intensive with other statements that give better performance.

Detailed information about the runtime analysis is available in the online documentation:

*ABAP Editor* → *Info Button* → *ABAP Overview* → *ABAP Tools* → *ABAP Analysis Tools*



# Appendix 3

## The Code Inspector

### Introduction

The *Code Inspector* lets you check your programs (or sets of programs) for performance, security, and typical semantic errors. Several of the check criteria are listed for each of these aspects below, to explain them in more detail:

#### Performance

- How are indexes used during database access?
- Have any SELECT statements been embedded in loops?

#### Security

- Are other clients accessed within the program?
- Are dynamic elements used in the SELECT statement?

#### Typical semantic errors

- Is the sy-subrc checked after each AUTHORITY CHECK statement?
- Is a client actually specified in the CLIENT SPECIFIED statement?
- Are several type E (error) messages sent consecutively?

### Calling the Code Inspector

There are several different ways of calling the *Code Inspector*.

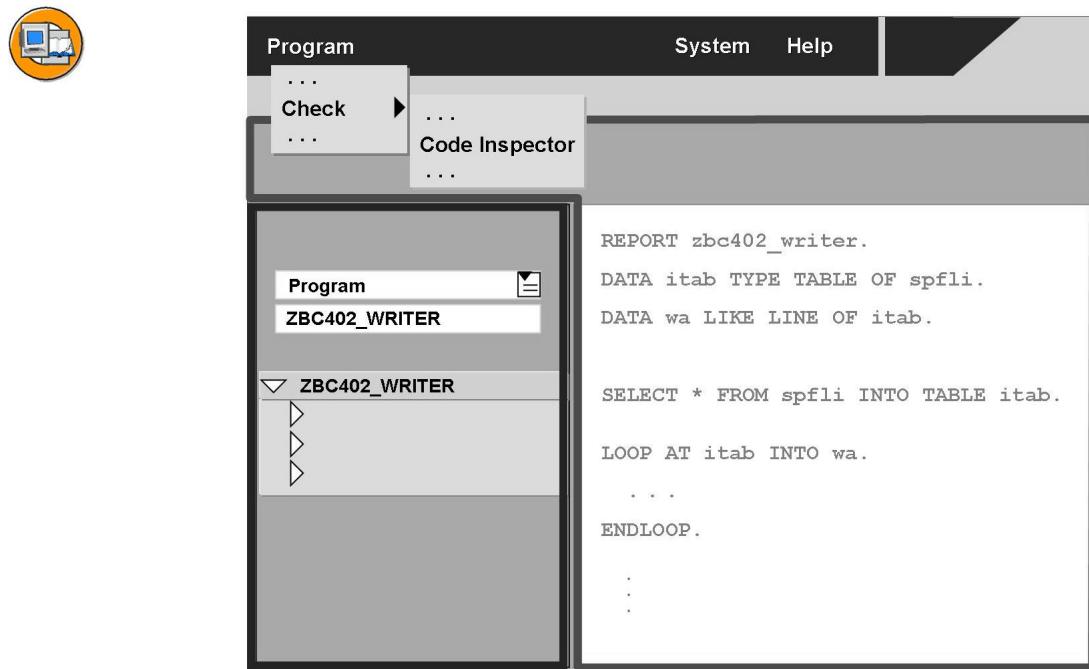


Figure 268: Direct Call of the Code Inspector from the ABAP Editor

The graphic shows one possibility. Choose the menu path *Program* → *Check* → *Code Inspector* to call the *Code Inspector* and start a standard inspection. You obtain the same result when you start the *Code Inspector* from the context menu in the object list.

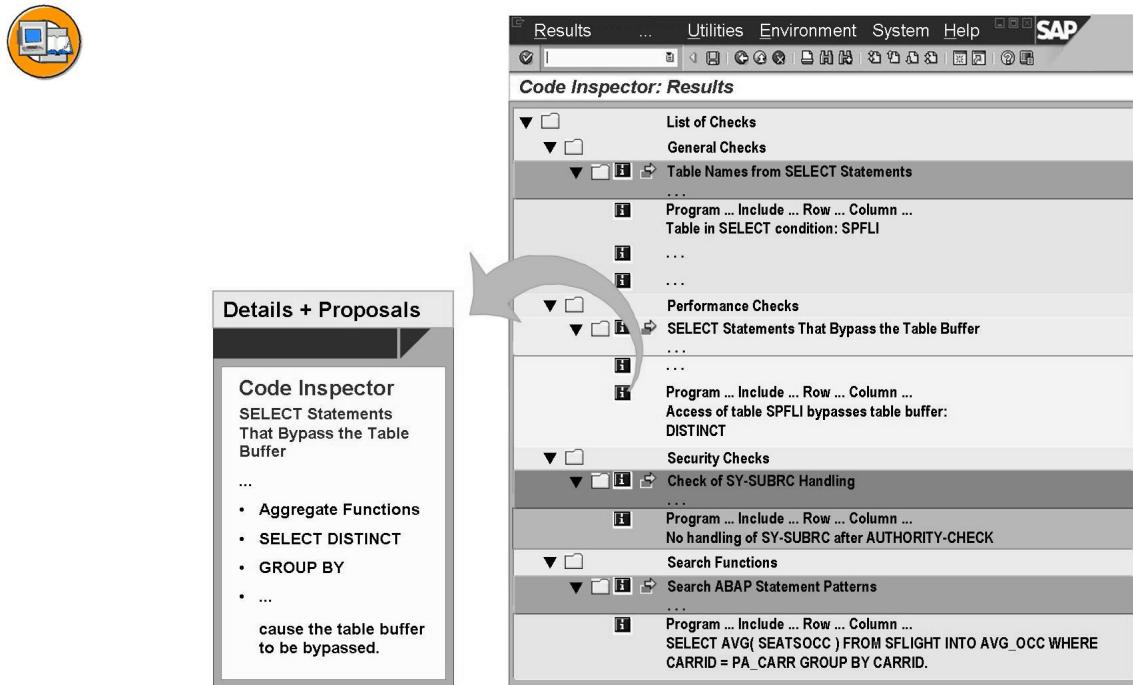


Figure 269: Inspection Results

As the result of an inspection you receive a list of error and warning messages. The **H** button that belongs to the message shows a detailed error description as well as suggestions for improvements. Double-clicking on the error text displays the corresponding program statement.

In this type of standard check, the *Code Inspector* uses a default check variant in which the checks to be performed are predefined. This default check variant contains:

- Extended syntax check
- Check of critical statements (such as C calls or Native SQL)
- Selected performance checks

You can overwrite this default check variant by creating a new check variant with the name **DEFAULT**. Note, however, that this new check variant overrides the standard variant for your user (in all clients).

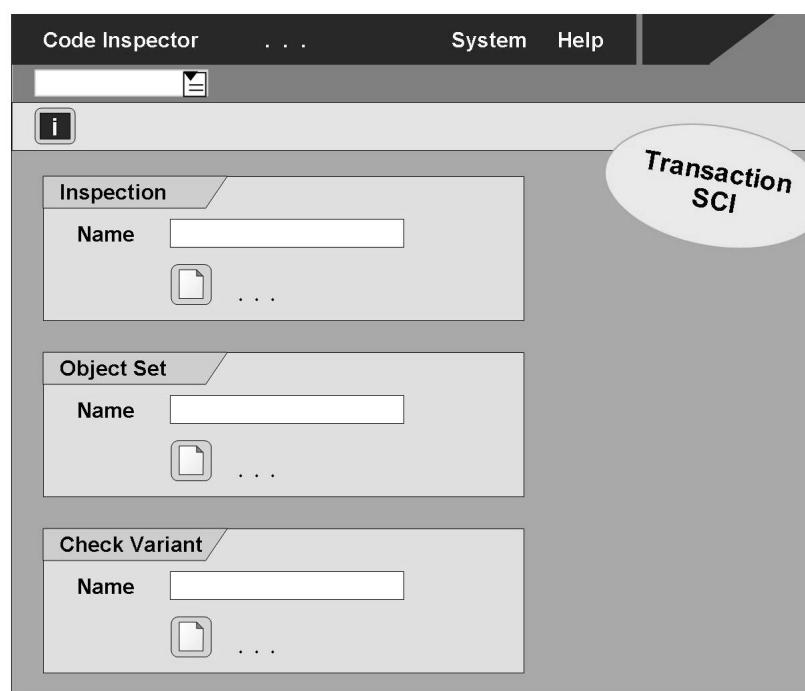
If you delete your default check variant, the standard variant is used automatically for future checks. For this reason, you should always define your own check variants instead of overwriting the default variant. We examine this option in the next section.



## Defining Inspections

To define custom checks, start the *Code Inspector* with transaction code SCI or follow the menu path *Tools* → *ABAP Workbench* → *Test* → *Code Inspector*. The initial screen contains three areas:

- Inspection
- Object Set
- Check Variant



**Figure 270: Defining Complex Inspections in Transaction SCI**

### Check Variant

Defines, which checks are to be performed

### Object set

This specifies what (Repository) objects should be checked.

### Inspection

Name of the actual check. If you want to save the inspection (persistent), enter a name; if you do not enter a name for the inspection, the results are not saved

Therefore, before you can start an inspection, you have to define an object set and a check variant.



**Hint:** You can define check variants, object sets, and inspections as either **public** or **private**. Use the pushbutton next to each input field to toggle between these two categories. Note that private objects are for your personal use only, whereas public objects are available to all users in the system.

Accordingly, the most important item when using the *Code Inspector* is the check variant. A check variant consists of one or more check categories, which in turn consist of one or more single checks (inspections). These single checks can be parameterized, for example, with a keyword or an indicator for a specific partial aspect of the check. In most cases, single checks only investigate a specific object type – such as the "Check of Table Attributes", which only examines DDIC tables.

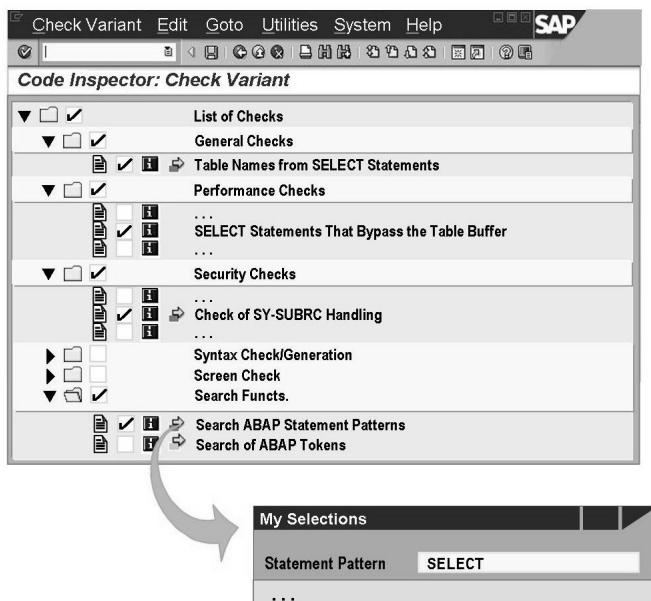


Figure 271: Creating a Check Variant

The single checks are assigned to various check categories. The most important check categories are described below:

- **General Checks** contain data formatting, such as the list of table names from SELECT statements
- **Performance Checks** contains checks of performance and resource usage, such as
  - Analysis of the WHERE condition for SELECT / UPDATE, and DELETE
  - SELECT statements that bypass the table buffer
  - Low-performance accesses of internal tables
- **Security Checks** contains checks of critical statements, cross-client queries, and insufficient authority checks
- **Syntax Check/Generation** contains the ABAP syntax check, an extended program check, and generation
- **Programming Conventions** contains checks of naming conventions
- **Search Functions** contains searches for tokens (words) and statements in ABAP coding

# Appendix 4

## SQL Trace

### Starting the SQL Trace

The SQL trace allows you to precisely analyze and track the database accesses that the database interface in the SAP system initiates. This makes the trace an effective tool for performance analysis.

To start the SQL performance trace, enter transaction code ST05 or choose menu path *System -> Utilities -> Performance Trace*.

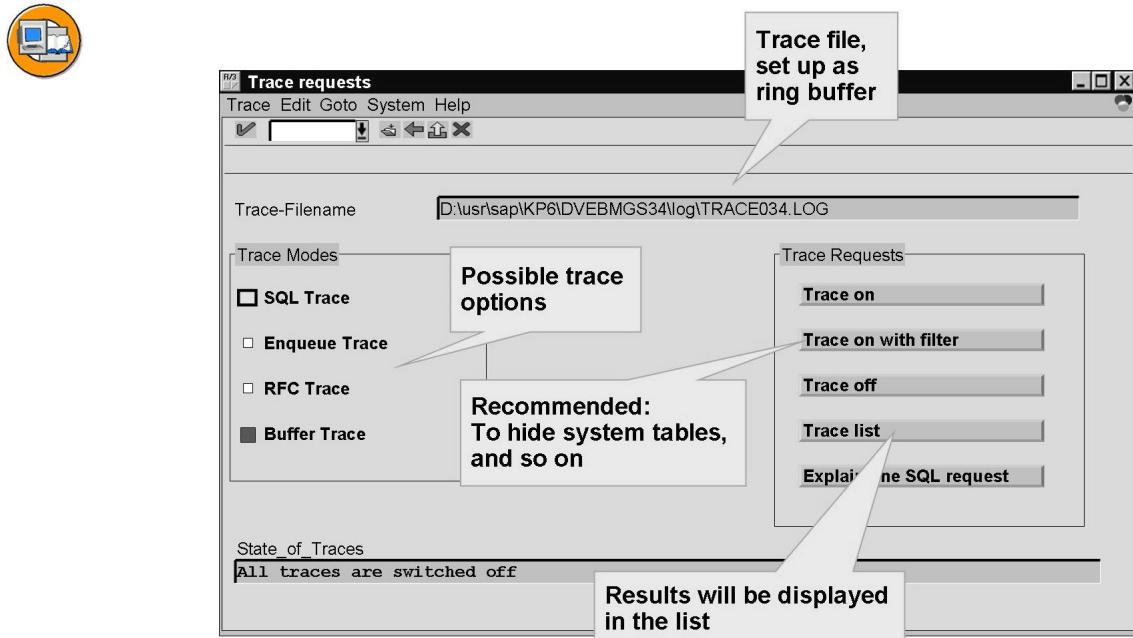


Figure 272: SQL Trace: Initial Screen and Overview

Before you activate the SQL trace for a program or transaction, you should execute that program or transaction once first, to avoid recording the refresh operations for the table buffer related to the SAP system, program buffer, and so on. You can activate the trace in a production system as well, because it does not cause any errors or inconsistencies. You can restrict the SQL performance trace to any user and any single object. You should make sure, however, that the user being recorded is not working in an additional session and that no other processing forms (batch processing, update tasks) are running in parallel under the same user on the same application server. If there are, the SQL performance trace is unreadable.



**Hint:** Each application server only has one active trace file. As a result, the trace can only be activated for **one user on each application server**. The standard size of the trace file is 800K. We recommend increasing the size of the trace file to 16 MB (parameter rstr/max\_filesize\_MB = 16 384 000).

## The SQL Trace List



Basic SQL List - Sorted by PID						
Trace Edit Goto System Help						
DDIC info Explain SQL ABAP display Extended						
Transaction =	PID =	Type =	RC	Statement	User	
Duration	Objectname	Oper	Rec			
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
805	VBAP	FETCH	5	1403		
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
785	VBAP	FETCH	5	1403		
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
788	VVBAP	FETCH	5	1403		
40	VBAK	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
1.061	VBAK	FETCH	10	1403		
1.327	VBAP	PREPARE	0	0	SELECT WHERE "MANDT" = :AO AND "VBELN"	
85	VBAP	OPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
6.919	VBAP	FETCH	49	1403		
826	VBAK_VBAP	PREPARE	0	0	SELECT WHERE "MANDT" = :AO AND "VBELN"	
41	VBAK_VBAP	OPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	
5.924	VBAK_VBAP	FETCH	49	1403		
800	VBAK	PREPARE	0	0	SELECT WHERE ( T_01 . "MANDT" = :AO A	
46	VBAK	OPEN	0	0	SELECT WHERE ( T_01 . "MANDT" = '100'	
5.704	VBAK	FETCH	49	1403		
217	VBAK	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL	

Figure 273: SQL Trace: Trace List

There are slight differences in the trace list between the different database systems. Data operations that take longer than 100 ms are highlighted in red. Each section in the trace list contains the user ID, transaction, SAP work process number, and client as header information. The columns in the expanded trace list have the following meanings:

**Duration**

Duration of the database operation in microseconds (depends on the system load)

**Object name**

Name of the table/ the view from the SQL statement

**Operation**

Database operation that is to be executed

**Records**

Number of data records that the database operation supplies

**RC (return code)**

Return code of the database system statement

**Statement**

Text of the observed SQL statement

When an SQL read statement is executed, the following DBSS operations take place:

The **PREPARE** operation parses the text of the SQL statement and translates it to a statement that the database can process. In particular, the access strategy is determined here. Bind variables (placeholders) are used for the variable values in the database statement. The **OPEN/REOPEN** operation replaces these bind variables in the database statement with specific values and the cursor is opened in the database. The actual transport of records from the database server to the application server takes place during the **FETCH** operation. The number of records transported by each **FETCH** operation is (partially) dependent on a profile parameter.

SQL write statements are processed similarly, with the exception that **EXEC/REEXEC** operations are performed instead of the **OPEN/REOPEN** and **FETCH** operations. DB cursors are buffered in the SAP cursor cache for each SAP work process, which makes it possible to skip the **PREPARE** operation when an SQL statement with an identical structure is executed.

## Analyzing SQL Statements Database-Specifically

The *Explain SQL* function lets you analyze individual SQL statements database-specifically.



Basic SQL List - Sorted by PID					
Duration	Object	Op.	Rec	RC	Statement
41	SBOOK	OPEN	0	0	SELECT WHERE "MANDT" = '100' AND . . .
805	SBOOK	FETCH	45	0	
245	SBOOK	FETCH	5	1403	



**"Explain SQL" returns the following information:**

- Display of the native SQL statement (*DB-specific*)
- Information about the optimizer strategy
- Information about the costs of the access (*DB-specific*)
- The index used by the statement
- Information about the table and its indexes
- Information about the optimizer statistics

**Figure 274: SQL Trace: EXPLAIN**

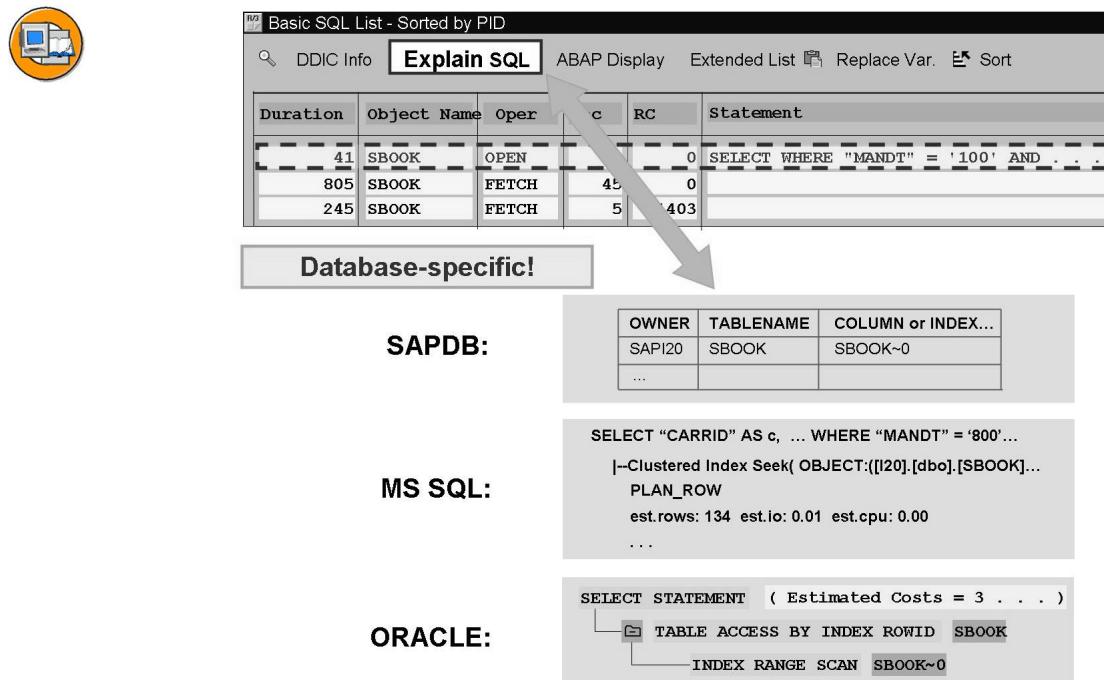
For the *Explain SQL* function, you exit the trace list by choosing the menu path *SQL Trace*→*Explain SQL* or by clicking the *Explain SQL* button.

*Explain SQL* is performed for the SQL statement in the current cursor position in the trace list.



**Hint:** *Explain SQL* is only possible for the following database operations: PREPARE, OPEN, and REOPEN. If you positioned the cursor on a FETCH operation, the Explain does not work.

Note that the use and functions of Explain SQL **vary widely between databases!**



**Figure 275: The Explain Is Database-Specific**

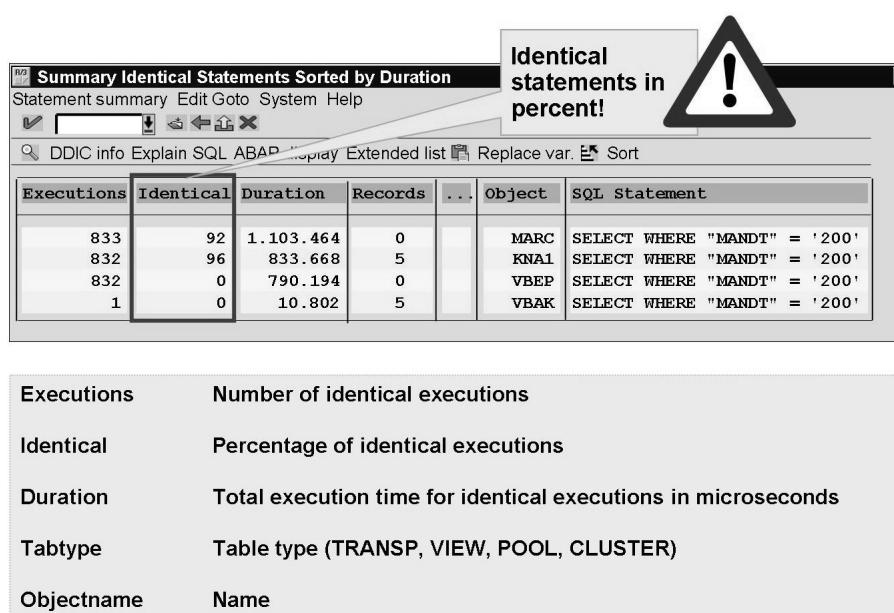
Some databases display the text of the SQL statement with placeholders (also called **bind variables**) in addition to the access strategy of the SQL statement. These variables serve as placeholders in case the same command is executed again with different contents. The proper navigation (usually a double-click) displays the content of the bind variables.

An important function within the Explain is the display of the **access strategy** that the optimizer uses – that is, the utilized index. The diagram shows examples for three databases. When a suitable index is used, the search effort required to compile the result set for the SQL statement can be reduced significantly. Therefore, developers can verify whether the resulting command was processed by the database quickly.

## Summarizing SQL Trace Lists

Options are available for summarizing the SQL trace list according to various criteria:

## Summarizing by SQL Statement



**Figure 276: SQL Trace: Sum of All Commands**

From within the trace list, choose menu path *Trace List -> Summarize Trace by SQL Statement* to display an overview of how many **identically structured SQL statements** were performed for the various tables and/or views. “Identically structured” statements use the same access strategy, possibly with different values for the bind variables.

The list is sorted by total time required in microseconds.

It contains the following information for each SQL statement:

- How often was the statement executed?
- How many records were transported (total and average)?
- How much time was required for the access (total and average)?
- Which percentage of the executions was completely identical to a previous execution – that is, with identical values in the bind variables?

The share of identical accesses is particularly relevant for performance aspects, because identical accesses usually return identical results. They can be avoided through data buffering.



**Hint:** In releases before SAP Web AS 6.10, select *Goto -> Statement Summary* to reach this point.

## Summarizing by Identical Access



Avoid identical selects!

Executions	Object Name	Where Clause
4	VBAK	"MANDT" = '100' AND "VBELN" BETWEEN '000000000
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000001'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000002'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000003'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000004'
2	VVBAP	"MANDT" = '100' AND "VBELN" = '0000000005'
2	VVBAP	"MANDT" = '100' AND "VBELN" = '0000000006'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000007'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000008'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000009'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000010'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000001' OR

Figure 277: SQL Trace: Identical SELECT Statements

A more detailed analysis of **identical SQL statements** is available under menu path *Trace List* → *Display Identical Selects* from within the trace list. “Identical” statements use the same access strategy with the same values for the bind variables.

Since identical SQL statements usually return identical result sets, they can often be avoided by buffering the results, which can significantly improve the performance of an application.

The optimization potential results from the identical executions of an SQL statement (for example: of the 4 identical executions of an SQL statement involving a database table, for example, 3 are unnecessary if the results are buffered in an internal table; the optimization potential for this SQL statement is therefore 75%).



**Hint:** In releases before SAP Web AS 6.10, select *Goto* → *Identical Selects* to reach this point.



# Index

## Numerics/Symbols

->\*, 237

## A

ABAP runtime system, 54  
 arithmetic  
 binary floating point, 129  
 decimal floating point,  
 131  
 integer, 128  
 mixed expression, 132  
 packed number, 130  
 assertion, 460  
 ASSIGN, 238  
 CASTING, 281  
 CASTING TYPE, 281

## B

Boxed Components, 80  
 Initial Value Sharing, 82  
 breakpoint, 460  
**C**  
 CALL TRANSACTION, 41  
 checkpoint, 460  
 checkpoint group, 465  
 COLLECT  
 ASSIGNING, 241  
 REFERENCE INTO, 241  
 conventions, 486

## D

data object  
 deep, 18  
 flat, 18  
 lifetime, 22  
 visibility, 20  
 data objects  
 definition, 13  
 data reference, 235

dereferencing operator,  
 237  
 reference semantics, 236  
 validity, 237  
 data references  
 generic, 284  
 type casting, 285  
 data types  
 complete, 15  
 generic, 275  
 incomplete, 15  
 numeric, 127  
 predefined, 13  
 predefined generic, 16

database, 364  
 indexes, 366  
 date & time fields, 167  
**DESCRIBE**  
 DESCRIBE FIELD, 304  
 DESCRIBE TABLE, 304  
 dynamic call, 261  
 dynamic OPEN SQL, 260  
 dynamic programming, 255  
 call executable program,  
 266  
 call function, 263  
 call method, 265  
 call modularization unit,  
 261  
 create objects, 329  
 dynamic OPEN SQL, 260  
 generate program, 266  
 token, 258

## F

field symbol, 237  
 validity, 238  
 field symbols  
 generic, 279  
 type casting, 281

- F**IELD-SYMBOLS, 238
- G**eneric data types, 275  
generically typed field symbols, 279  
generically typed parameters, 278
- I**ncludes, 7  
inner join, 410
- INSERT** ASSIGNING, 241  
REFERENCE INTO, 241
- internal tables, 189  
aggregation, 217  
binary search, 213  
data reference, 239  
delete duplicates, 215  
field symbol, 239  
hashed, 194  
index & key accesses, 195  
secondary key, 218  
sorted, 192  
standard, 191
- IS ASSIGNED**, 238
- IS BOUND**, 237
- IS INITIAL**, 237
- L**EAVE TO TRANSACTION, 41  
literals, 19
- LOAD**, 56  
locators, 452  
logical operators, 165  
logpoint, 460
- LOOP AT** ASSIGNING, 240  
REFERENCE INTO, 240
- M**emory management  
ABAP memory, 72  
Boxed Components, 80  
CALL FUNCTION, 66  
CALL METHOD, 66
- C**ALL TRANSACTION, 62  
external session, 60  
internal session, 60  
internal table, 78  
**LEAVE TO TRANSACTION**, 62  
**PERFORM**, 70  
program group, 61  
PXA buffer, 58  
roll area, 58  
SAP memory, 72  
shared objects, 94  
string, 77  
**SUBMIT**, 62
- MODIFY** ASSIGNING, 241  
REFERENCE INTO, 241
- N**aming conventions, 486  
null reference, 237
- O**pen SQL  
locators, 452  
performance rules, 501  
SELECT, 25  
streams, 449  
table buffer, 371
- OPEN SQL**  
dynamic, 260  
outer join, 410
- P**erformance hints, 491  
program  
activate, 57  
declarations, 5  
events, 6  
generate, 55  
includes, 7  
introductory statements, 4  
processing blocks, 5  
structure, 3
- Program  
**LOAD**, 56

- program groups
  - additional program group, 66
  - main program group, 66
- programming errors, 480
- R**
- READ TABLE
  - ASSIGNING, 240
  - REFERENCE INTO, 240
- recommendations, 488
- regular expressions, 156
- RTTC, 334
- RTTI, 304
- Runtime Analysis, 519
- runtime type creation, 334
- runtime type identification, 304
- S**
- SELECT, 25
  - aggregate functions, 393
  - array fetch, 385
  - DISTINCT, 392
  - ENDSELECT, 27
  - FOR ALL ENTRIES, 418
  - GROUP BY, 397
  - INNER JOIN, 410
  - INTO TABLE, 28
  - multiple datasets, 385
  - nested, 409
  - ORDER BY, 390
  - OUTER JOIN, 410
  - PACKAGE SIZE, 386
  - SINGLE, 26
- single dataset, 384
- subquery, 413
- using cursor, 387, 415
- WHERE, 380
- shared objects, 94
- streams, 449
- string expressions, 153
- string functions, 149
- string operations, 146
  - byte / character type, 148
  - date & time fields, 167
  - expressions, 153
  - functions, 149
  - logical operators, 165
  - regular expressions, 156
  - statements, 148
  - subfield access, 165
  - templates, 153
  - unicode, 147
- string templates, 153
- SUBMIT, 39
- subquery, 413
- synchronous program call, 37
  - CALL TRANSACTION, 41
  - LEAVE TO TRANSACTION, 41
  - SUBMIT, 39
- T**
- token, 258
- U**
- UNASSIGN, 238



# Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.