BC100

Introduction to Programming (Based on ABAP Examples)

SAP NetWeaver Application Server - ABAP

Date _____

Training Center _____

Instructors _____

_____

Education Website _____

Participant Handbook

Course Version: 91
Course Duration: 2 Day(s)
Material Number: 50094355

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

Internal Use SAP Partner Only

Copyright

Trademarks

Disclaimer

Internal Use SAP Partner Only

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
Example text	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.
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
`Example text`	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Internal Use SAP Partner Only

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

Internal Use SAP Partner Only

Course Overview

This course provides an introduction to programming based on examples from SAP's programming language, ABAP. You are introduced to the ABAP Editor (a development tool) and learn how to develop simple programs. In both cases, the focus is placed on the concepts and fundamental principles. You are also introduced to the relevant terminology, allowing you to understand the in-depth documentation more easily.

These topics are covered using practical examples, so that you can immediately implement what you have learned. In particular, it should become apparent how little effort is needed to develop programs in the ABAP Editor.

By attending this course, participants build up the knowledge they require to attend the course BC400 (ABAP Workbench Foundations), allowing them to concentrate on the essentials in the follow-on course.

This course is therefore a prerequisite for all new programmers to attend BC400.

Target Audience

This course is intended for the following audiences:

- Project leaders
- Project team members
- Developers
- Consultants

Course Prerequisites

Required Knowledge

- None

Course Goals

This course will prepare you to:

- Familiarize yourself with basic programming concepts even if you have no or little previous knowledge of programming
- Build up the programming knowledge required to attend BC400

Course Objectives

After completing this course, you will be able to:

Internal Use SAP Partner Only

- Describe the basic concepts of programming and implement what you have learned in simple ABAP programs

Unit 1

From the Problem to the Program - Introduction to Structured Programming

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Identify problems
- Devise system-independent solutions to these problems
- Implement solutions to problems in program flowcharts (PFCs)
- Implement solutions to problems in structograms

Unit Contents

Lesson: From the Problem to the Algorithm

Lesson Overview

This lesson introduces you to the term “algorithm”. You will identify a problem and devise a solution to this problem.

Lesson Objectives

After completing this lesson, you will be able to:

- Identify problems
- Devise system-independent solutions to these problems

Business Example

An employee from the company "Calculate & Smile" is to become a programmer in the ABAP environment. Before attending the course BC400 (ABAP Workbench Foundations), the employee needs to acquire basic programming knowledge. The employee is to identify a problem and devise a suitable system-independent solution (an algorithm). This is an integral part of the development process and must be completed before programming begins.

Problem

A problem (an unsatisfactory situation) that needs to be solved is also referred to as a task

Figure 1: Problem

A problem (an unsatisfactory situation) that needs to be solved is also referred to as a task. The path to devising a solution to this task often requires overcoming some obstacles. A way of transforming an initial situation (current state) into a goal (target state) needs to be found. Finding a solution to a task often requires using existing knowledge or breaking the task down into subtasks that can later be divided into more detailed units.

Problem Solving

Problem solving is an activity carried out to find a solution to a task

Figure 2: Problem Solving

Problem solving is an activity carried out to find a solution to a task. The approach taken to find a solution can be heuristic (learning by trial and error) or can involve applying a solution strategy. In computer science, a solution strategy is referred to as an algorithm. Provided an intelligent being can gather information (that can potentially help solve a problem), it can develop a solution strategy. If no information is available, the intelligent being carries out various activities (depending on the problem) until a solution is found or efforts are abandoned. If a solution is found, powers of recollection allow it to be reapplied again later if the situation reoccurs. This means that all activities do not have to be repeated. The time available and resources required to find a solution also play a crucial role in the process.

Steps to Solving a Problem

Solving problems involves three main steps:

1. Identifying the problem (current state)
2. Devising and implementing a solution or algorithm
3. Checking the results (target state)

Figure 3: Steps to Solving a Problem

When a problem needs to be solved, it must first be understood before solutions are sought. If the steps required to reach a solution are defined explicitly, this is referred to as an algorithm.

Solving problems involves three main steps:

1. Identifying the problem (current state)
2. Devising and implementing a solution or algorithm
3. Checking the results (target state)

Solving Problems in Substeps

1. Identifying the problem and recording the current state
2. Gathering information that may help solve the problem (aids, solution approaches, and so on)
3. Selecting, grouping, and consolidating this information
4. Reducing the information to the information required
5. Evaluating the information
6. Applying solution strategies to devise solutions
7. Considering opportunities and risks
8. Solving the problem
9. Checking the result (target state)

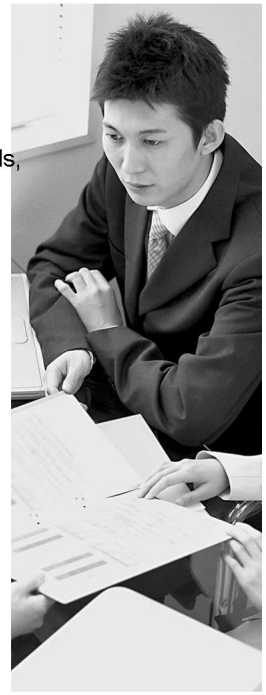


Figure 4: Solving Problems in Substeps

These steps can be broken down into the following substeps:

1. Identifying the problem and recording the current state
2. Gathering information that may help solve the problem (aids, solution approaches, and so on)
3. Selecting, grouping, and consolidating this information
4. Reducing the information to the information required
5. Evaluating the information
6. Applying solution strategies to devise solutions
7. Considering opportunities and risks
8. Solving the problem
9. Checking the results (target state)

The following example is intended to explain the procedure:

Substep 1: The CD "No Problemo" needs to be located from a CD stand within 30 seconds.

Substep 2 to 4: The location of the "No Problemo" CD in the CD stand is unknown. There are a total of 100 CDs in the stand. They are arranged in alphabetical order. To locate the required CD, you could use one of the following approaches:

- Perform a linear search through all CDs (one after another) from top to bottom.
- Perform a linear search through all CDs from bottom to top.
- At a particular location in the CD stand, take out a CD. Based on the name of the CD, perform a linear search up or down the stand.
- Perform a linear search through the CDs and assign a number to each CD. Record the numbers in a list, allowing you to find CDs faster in future based on the numbers assigned. This approach could also be used even if the CDs are not sorted in the stand.
- Take out the CD located exactly half way down the stand. Depending on whether the title of the CD comes earlier or later in the alphabet, divide the relevant half of the stand and repeat the same check. These steps are repeated until the correct CD is found or it is evident that the CD is not on the stand. This type of search is known as a binary search.
- The following solution is similar to the previous approach. Assuming the CDs are sorted alphabetically, it can be presumed that CD titles (where they are used) beginning with "A" are at the top, and CD titles beginning with "Z" are at the bottom. This approach aims to start the search at a location closer to the required CD than the center of the stand, based on the title of the CD. The alphabet has 26 letters. If you divide the number of CDs by the number of letters in the alphabet, and multiply the result with the ordinal location of the letter (with which the CD title begins) in the alphabet, the result is generally a more favorable starting point for the search. The algorithm for the binary search is then used. The search applied here is known as an interpolation search.
- ...

Substep 5: All these solutions identified are viable. However, the first four approaches are particularly time-intensive. Once the CDs are assigned numbers, (approach 4) it is possible to locate the CD directly, which guarantees the fastest search. Approach 5 always yields a particularly fast result. The interpolation search (approach 6) generally always provides the fastest result.

Substep 6: The approach based on the list of titles and the reference to a number, or locating the CD using the search algorithm (approach 5), are favored:

- Write a list of the titles of all CDs and assign a number to each CD. Attach a label indicating the CD number to the outside of each CD. Any new CDs added are to be entered in the list and assigned the next available number.
- Divide the total number of CDs by two and save the value in a storage medium (in a variable named DISTANCE). This value is assigned to a second storage medium (variable POSITION). The CD located at POSITION is taken from the stand. If the title of this CD comes before the title of the required CD in the alphabet, the DISTANCE variable is divided by two and the value calculated is subtracted from the value of the POSITION variable. The CD at this new position is taken from the stand and the title is checked again. If the title of this CD comes after the required title in the alphabet, the DISTANCE variable is divided by two and the value calculated is added to the value of the POSITION variable. This process is repeated. To achieve an accurate result, it must also be taken into account that the distance values calculated have to be rounded up or down.

Substep 7: Approach 4 to the solution (creating a list) demands that the list be updated each time a CD is added to or removed from the collection. You need to decide whether the CDs are to be added to the bottom and assigned a new number. Perhaps the owner of the CDs would prefer to arrange all CDs from a particular band beside one another rather than adding them to the bottom. Approach 5 (binary search) or approach 6 (interpolation search) would not require any maintenance effort. New CDs are inserted at the correct location based on alphabetical sorting.

Substep 8: The preferred approach is implemented in pseudo code:

1. Alphabetically sort the 100 CDs in the CD stand according to band and title.
2. Specify the band and title of the required CD.
3. Calculate at which position in the alphabet the initial letter of the band occurs.
4. Calculate the initial search position: number of CDs divided by 26 (number of letters in the alphabet) * the ordinal location of the letter in the alphabet.
5. Execute and repeat the following steps until the required CD is found or it can be deduced that the CD being searched for is not on the stand:
   - Take the CD located at the position calculated from the stand.
   - Do the band name and title of this CD come after those of the required CD in the alphabet? If so, a value that is calculated (distance) is subtracted from the current value (current position). If the title found occurs before the title being searched for in the alphabet, a specific value (distance) is added to the current position.
   - If the distance calculated is less than or equal to zero, the CD cannot be found and the search can be aborted.

Substep 9: Check whether the solution (algorithm) works and whether it can be applied in practice.

Exercise 1: Developing an Algorithm

Exercise Objectives

After completing this exercise, you will be able to:

- Develop an algorithm

Business Example

You work for the company "Calculate & Smile" and are to become a programmer in the ABAP environment. Before attending the course BC400 (ABAP Workbench Foundations), you need to acquire basic programming knowledge. You need to identify a problem and devise a suitable system-independent solution (an algorithm). This is an integral part of the development process and must be completed before programming begins.

Task 1:

Make various calculations.

1. You want to use a calculator to add, subtract, multiply, and divide two numbers. Describe the steps you follow to reach your solution. (The calculator is already switched on and ready for use.)

Task 2:

Engage in a telephone conversation.

1. You want to make a call. Engage in a complete telephone conversation from start to finish. Remember that you may also receive a call. Also bear in mind that the call may take place on a landline phone (with a handset) or a cell phone.

Solution 1: Developing an Algorithm

Task 1:

Make various calculations.

1. You want to use a calculator to add, subtract, multiply, and divide two numbers. Describe the steps you follow to reach your solution. (The calculator is already switched on and ready for use.)

   a) Sample solution for the addition

      1. Enter the first number.
      2. Press the “+” button.
      3. Enter the second number.
      4. Press the “=” button.
      5. The result is displayed.

Continued on next page

Internal Use SAP Partner Only

Task 2:

Engage in a telephone conversation.

1. You want to make a call. Engage in a complete telephone conversation from start to finish. Remember that you may also receive a call. Also bear in mind that the call may take place on a landline phone (with a handset) or a cell phone.

   a) The following example outlines a possible series of events for the call. There are of course other alternative solutions.

Is the phone ringing?
Yes: Is the cell phone ringing?
Yes: Press the button to take the call and start the conversation. After the call has ended, press the key to hang up.
No: Pick up the handset for the landline phone and start the conversation. After the call has ended, return the handset to its cradle.
No: Is the cell phone to be used for the call?
Yes: Is the number known?
Yes: Is the keypad on the cell phone locked?
Yes: Unlock the keypad.
No:
Enter the number (including the dialing code).
No: Is the number already saved in the internal memory of the cell phone?
Yes: Select the number.
No: Search for the number in the phone book (or Internet) and enter it.
Press the button to make the call. When the call had ended, press the button to hang up.
No: …

Figure 5: Example: Telephone Call

Lesson Summary

You should now be able to:

- Identify problems
- Devise system-independent solutions to these problems

Internal Use SAP Partner Only

Lesson: From the Algorithm to the Program Flowchart

Lesson Overview

Now that you have compiled a solution (an algorithm) for a problem, this lesson shows you how to depict your solution graphically in a program flowchart.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement solutions to problems in program flowcharts (PFCs)

Business Example

As an employee of "Calculate & Smile", you have already devised an algorithm, and now want to learn how to represent your solution in a graphic. The program flowchart is used to convert an algorithm into a program and outlines the sequence of steps required to solve a task.

From the Algorithm to the Program Flowchart

Once you have solved the problem and created an algorithm, you need to outline the individual solution steps graphically in a program flowchart (PFC). The program flowchart is a flow diagram that can be used as a starting point for computer programs. Symbols are used to display the flow graphically. These symbols are standardized in DIN 66001. Program flowcharts can also be used to represent processes that have nothing to do with computer programs. For this reason, no programming-language-specific commands are to be used. Program flowcharts start and end with an oval. The direction of the program flow is indicated by an arrow. The following figures introduce some of the symbols used. Information on all symbols is available in DIN 66001.

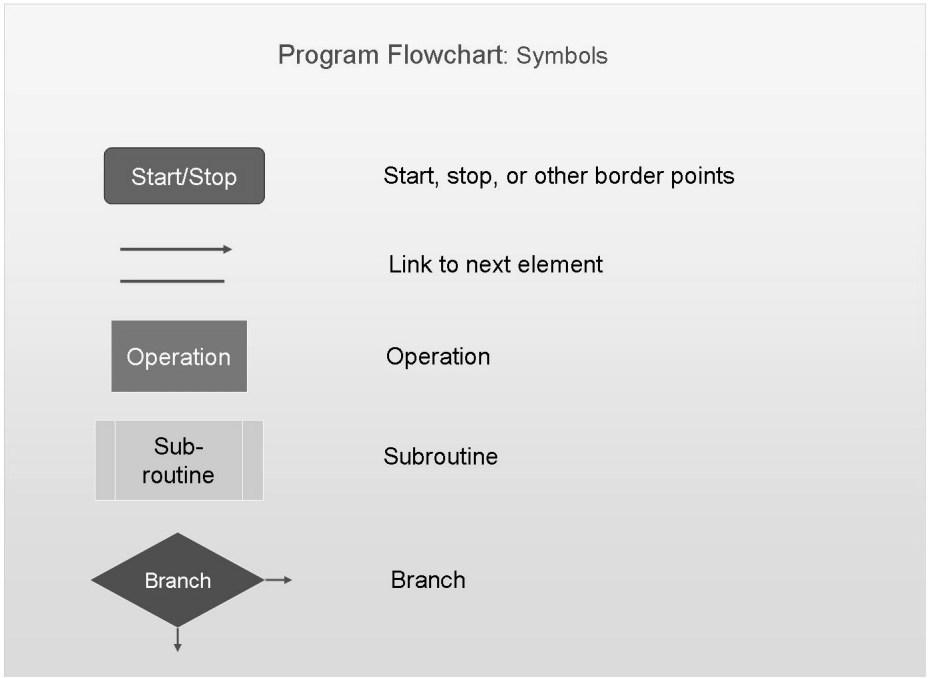


Figure 6: Symbols Part 1

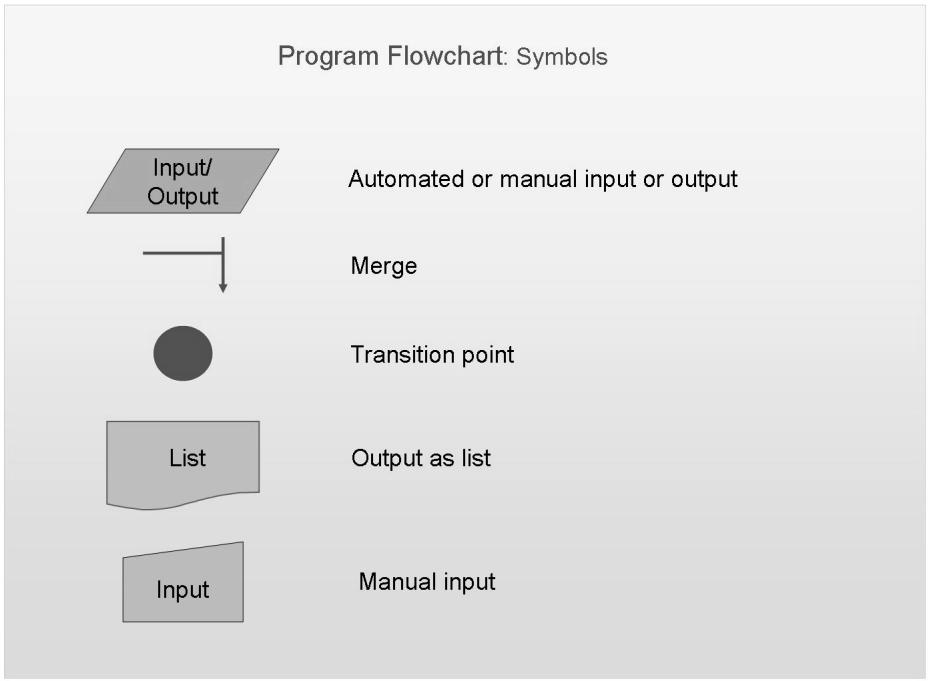


Figure 7: Symbols Part 2

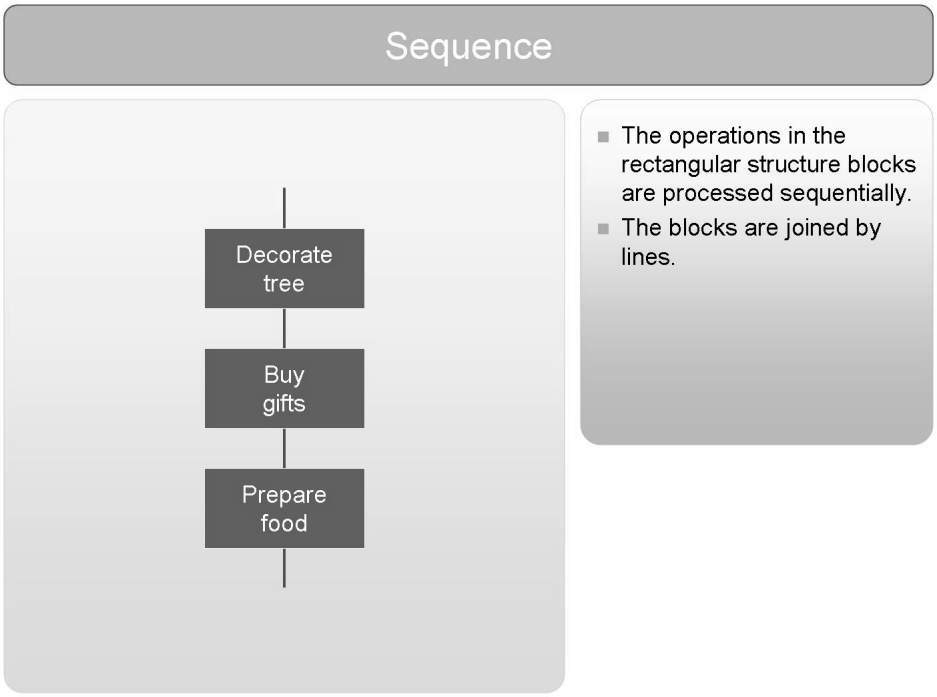


Figure 8: Sequence

Each operation is entered in a separate structure block. Each of the structure blocks are connected by lines and are arranged from top to bottom in the sequence they are processed.

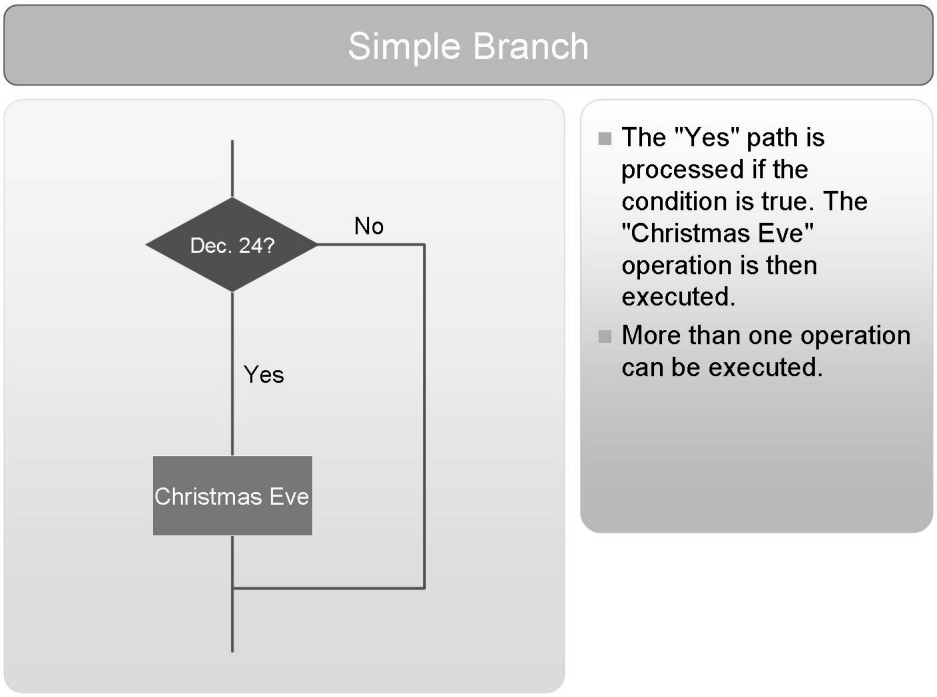


Figure 9: Branch

This branch consists of one condition and a section of code. It determines whether this program step is executed based on a condition. More than one operation can be executed.

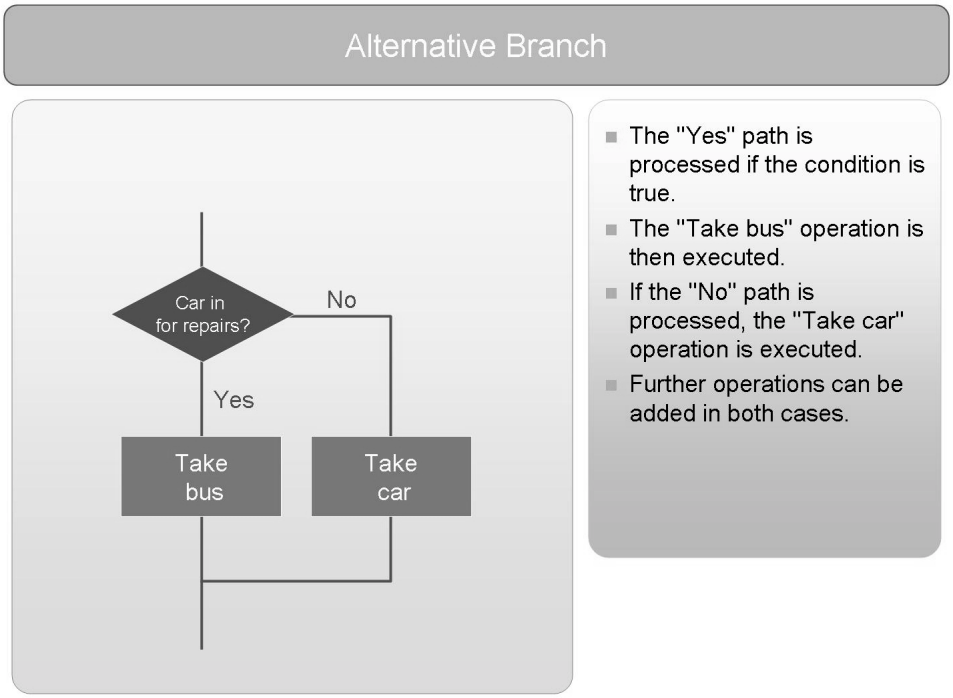


Figure 10: Alternative Branch

This is a branch that consists of one condition and two (or more) sections of code. It determines which of the program steps is executed based on a condition.

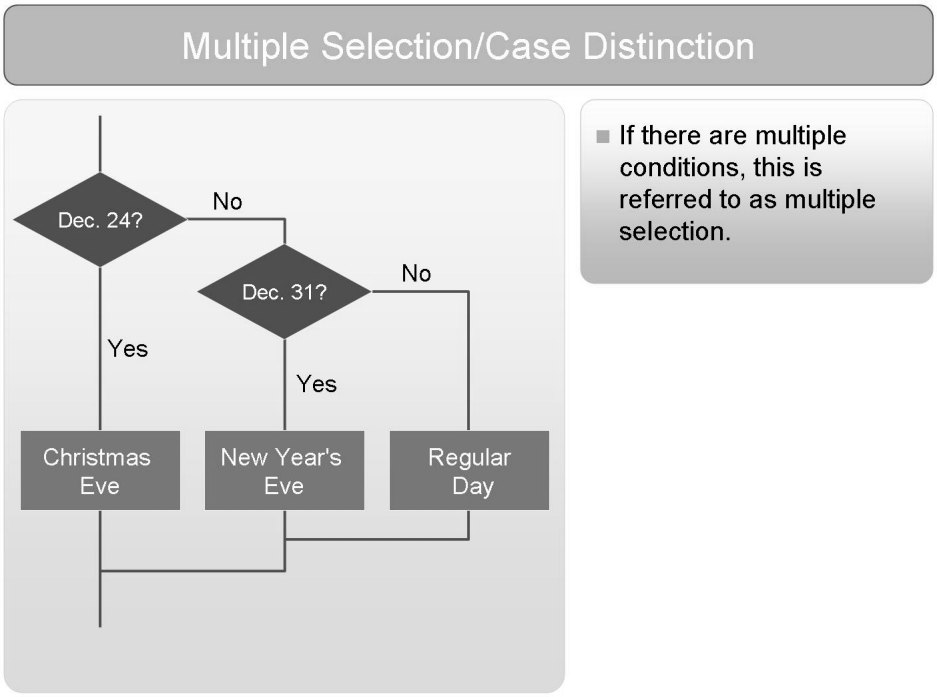


Figure 11: Multiple Selection/Case Distinction

With multiple selection, which is also referred to as case distinction, conditions are evaluated in sequence. If a condition is true, the corresponding section of code is processed. Otherwise the "No" path is taken.

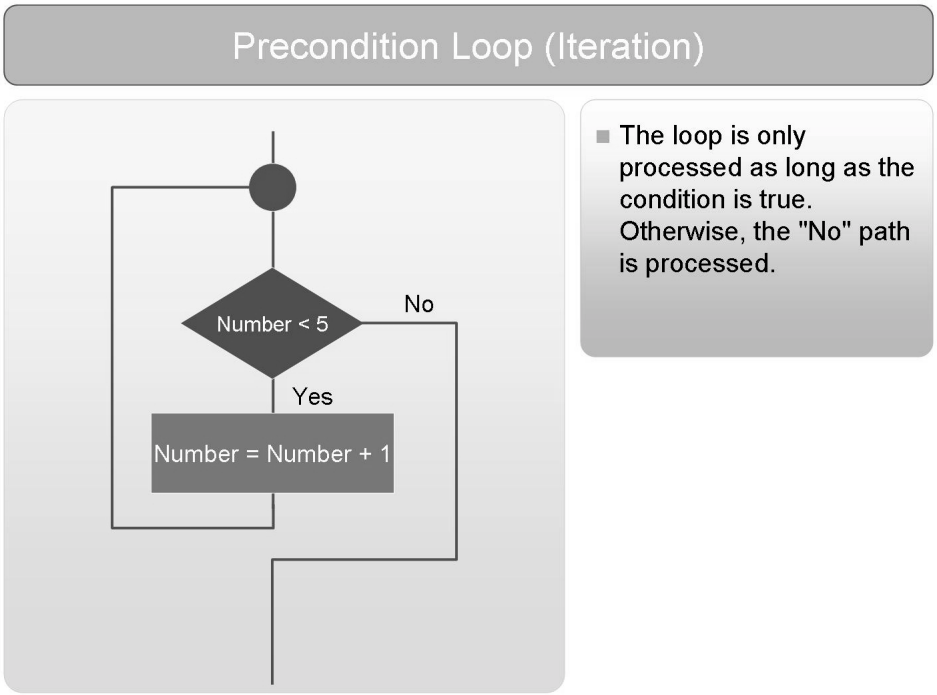


Figure 12: Precondition Loop

Within a loop, operations are repeated as long as the loop condition is met or until the loop is exited prematurely. If the condition is always true, this results in an infinite loop. In a precondition loop, a check is first made to establish whether a condition is met. If it is not met, the loop body is not processed.

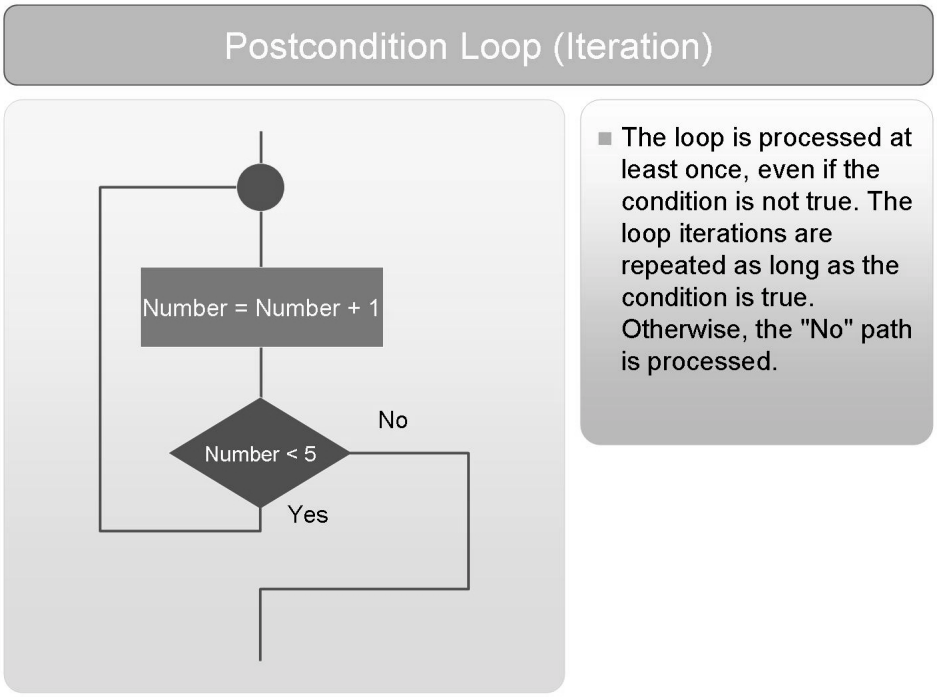


Figure 13: Postcondition Loop

In a postcondition loop, the loop body is first processed and then the condition is checked. If the condition is false, the loop is exited. Otherwise, another loop pass follows.

Exercise 2: Developing a Program Flowchart (PFC)

Exercise Objectives

After completing this exercise, you will be able to:

- Represent a solution (an algorithm) in a program flowchart

Business Example

After the "Calculate & Smile" employee has devised an algorithm, he or she now wants to learn how to depict the solution in a graphic. The program flowchart is used to convert an algorithm into a program and outlines the sequence of steps required to solve a task.

Task:

Create a program flowchart.

1. Create a program flowchart for a calculator that is to make calculations based on the “+”, “-”, “*”, and “/” operators. Note that dividing by zero results in an error and that this is to be indicated by displaying “Division by Zero” on the calculator display. If no operator is entered, “Incorrect Operator” is to be displayed.

Internal Use SAP Partner Only

Solution 2: Developing a Program Flowchart (PFC)

Task:

Create a program flowchart.

1. Create a program flowchart for a calculator that is to make calculations based on the “+”, “-”, “*”, and “/” operators. Note that dividing by zero results in an error and that this is to be indicated by displaying “Division by Zero” on the calculator display. If no operator is entered, “Incorrect Operator” is to be displayed.

   a) See result:

Result

Sample solution:

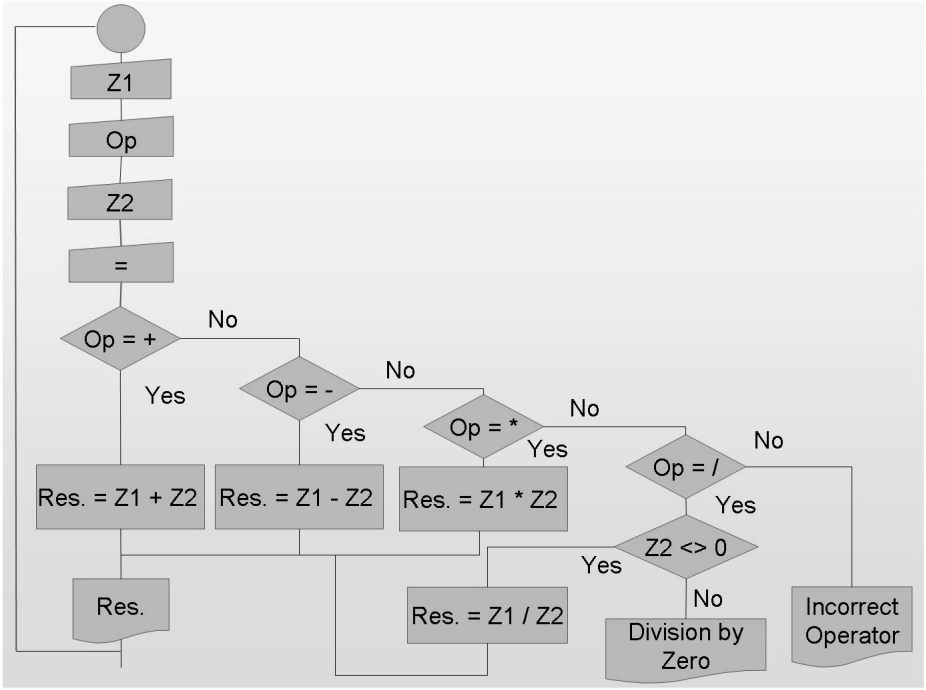


Figure 14: Solution: Program Flowchart

Lesson Summary

You should now be able to:

- Implement solutions to problems in program flowcharts (PFCs)

Lesson: From the Algorithm to the Structogram

Lesson Overview

Now that you have compiled a solution (an algorithm) for a problem, this lesson shows you how to depict your solution graphically in a structogram.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement solutions to problems in structograms

Business Example

Using the Nassi-Shneiderman diagram (structogram), the problem you want to solve as the "Calculate & Smile" employee using the algorithm already devised is reduced to smaller and smaller units until only sequences and control flow constructs remain. Using this type of diagram makes it possible to develop programs in a structured way, and the results can be implemented in many programming languages.

From the Algorithm to the Structogram

A structogram represents another way of depicting a solution (algorithm) graphically. The structogram is a flow diagram that can be used as a starting point for computer programs. Symbols are used to display the flow graphically. These symbols are standardized in DIN 66261. Structograms can also be used to represent processes that have nothing to do with computer programs. For this reason, no programming-language-specific commands are to be used. Each operation is written in a separate structure block, even if these operations occur more than once. A structogram is processed from top to bottom. The following figures introduce some of the symbols used. Information on all symbols is available in DIN 66261.

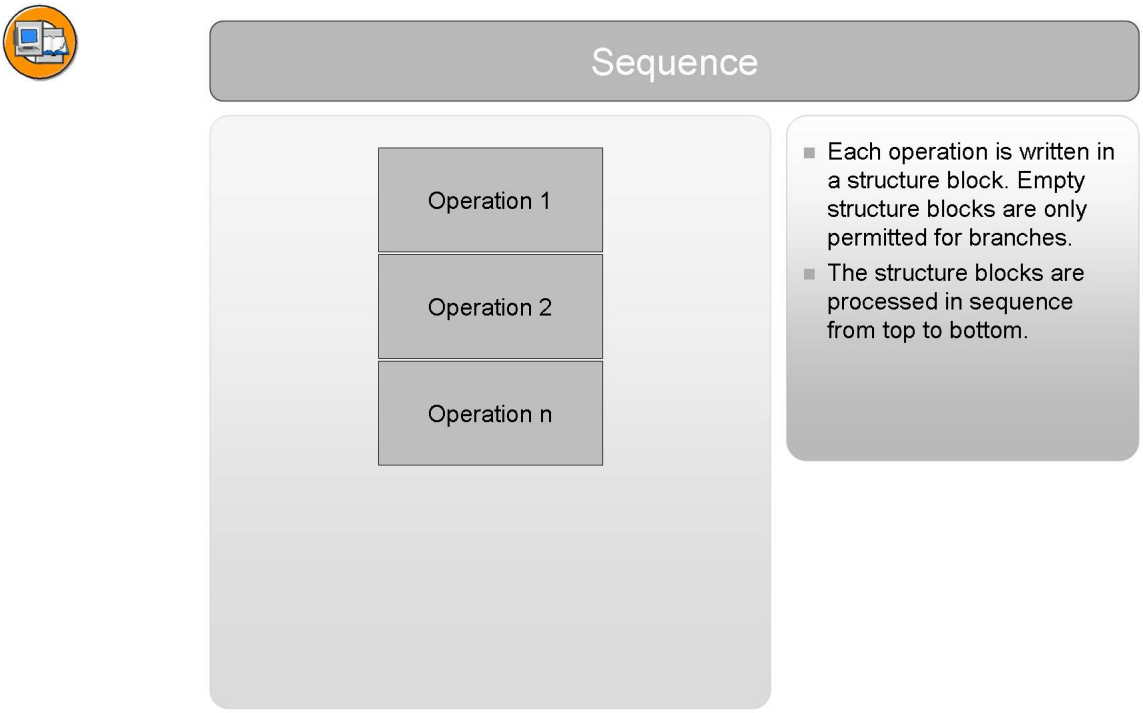


Figure 15: Sequence

Each operation is entered in a separate structure block. The individual structure blocks are processed in sequence from top to bottom.

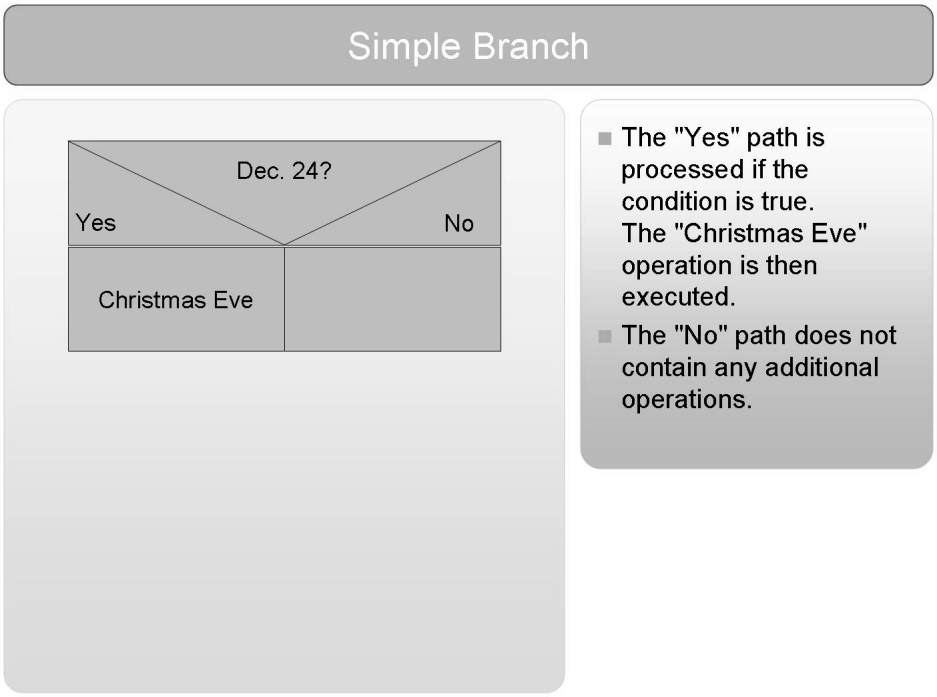


Figure 16: Simple Branch

This branch consists of one condition and a section of code. It determines whether this program step is executed based on a condition. More than one operation can be executed.

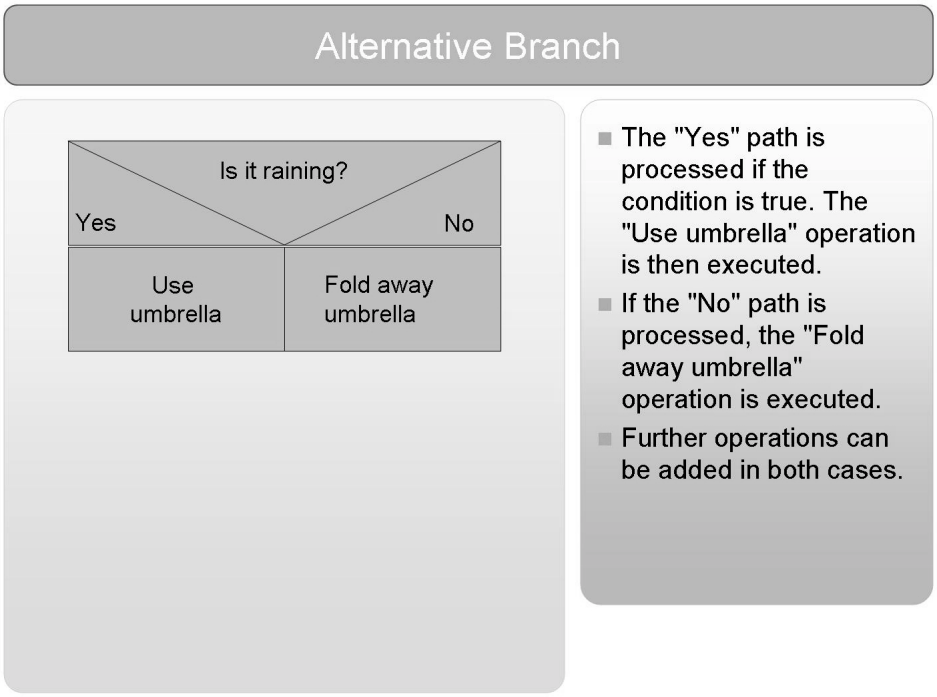


Figure 17: Alternative Branch

This is a branch that consists of one condition and two (or more) sections of code. It determines which of the program steps is executed based on a condition.

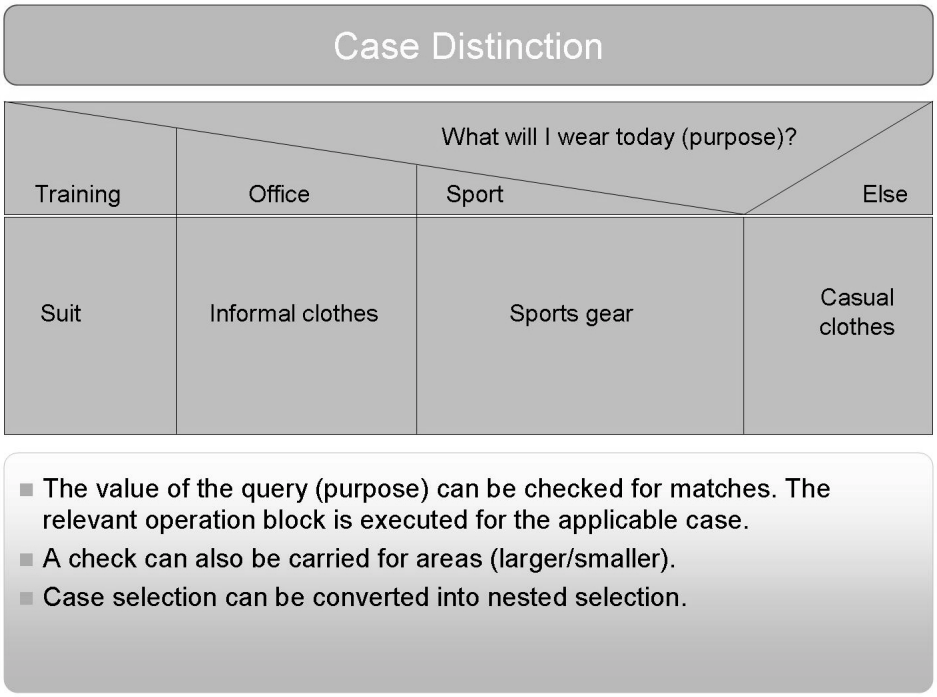


Figure 18: Multiple Selection/Case Distinction

With multiple selection, which is also referred to as case distinction, conditions are evaluated in sequence. If a condition is true, the corresponding section of code is processed. Otherwise the “Else” path is taken.

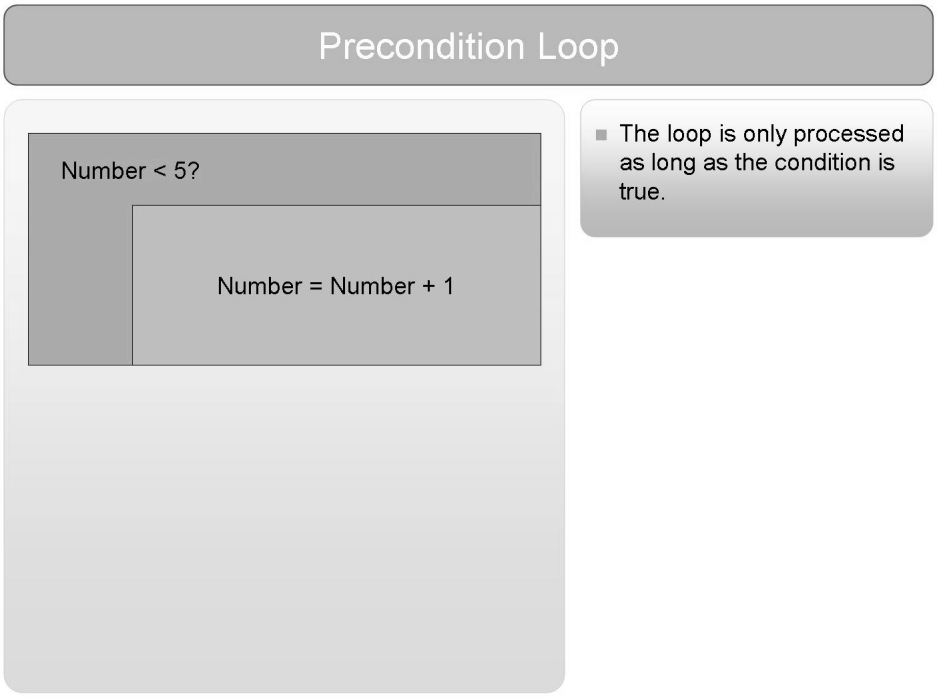


Figure 19: Precondition Loop

Within a loop, operations are repeated as long as the loop condition is met or until the loop is exited prematurely. If the condition is always true, this results in an infinite loop. In a precondition loop, a check is first made to establish whether a condition is met. If it is not met, the loop body is not processed.

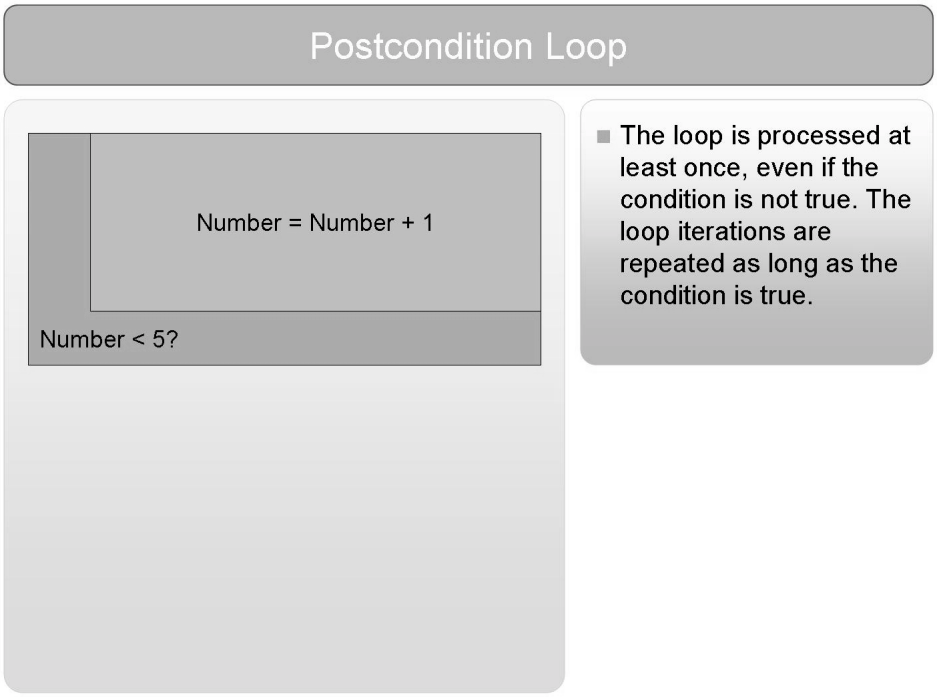


Figure 20: Postcondition Loop

In a postcondition loop, the loop body is first processed and then the condition is checked. If the condition is false, the loop is exited. Otherwise, another loop pass follows.

Exercise 3: Developing a Structogram

Exercise Objectives

After completing this exercise, you will be able to:

- Present a solution (an algorithm) in a structogram

Business Example

Using the Nassi-Shneiderman diagram (structogram), the problem you want to solve as the "Calculate & Smile" employee using the algorithm already devised is reduced to smaller and smaller units until only sequences and control flow constructs remain. Using this type of diagram makes it possible to develop programs in a structured way, and the results can be implemented in many programming languages.

Task:

Create a structogram.

1. Create a structogram for a calculator that is to make calculations based on the “+”, “-”, “*”, and “/” operators. Note that dividing by zero results in an error and that this is to be indicated by displaying “Division by Zero” on the calculator display. If no operator is entered, “Incorrect Operator” is to be displayed.

2. If you have time:

   Create a structogram for the “making a call” requirement. Remember that there are cordless phones and phones with handsets. Some phones have functions to save numbers, while others do not. Decide whether you are to take a call or make a call yourself.

Solution 3: Developing a Structogram

Task:

Create a structogram.

1. Create a structogram for a calculator that is to make calculations based on the “+”, “-”, “*”, and “/” operators. Note that dividing by zero results in an error and that this is to be indicated by displaying “Division by Zero” on the calculator display. If no operator is entered, “Incorrect Operator” is to be displayed.

   a) Structogram:

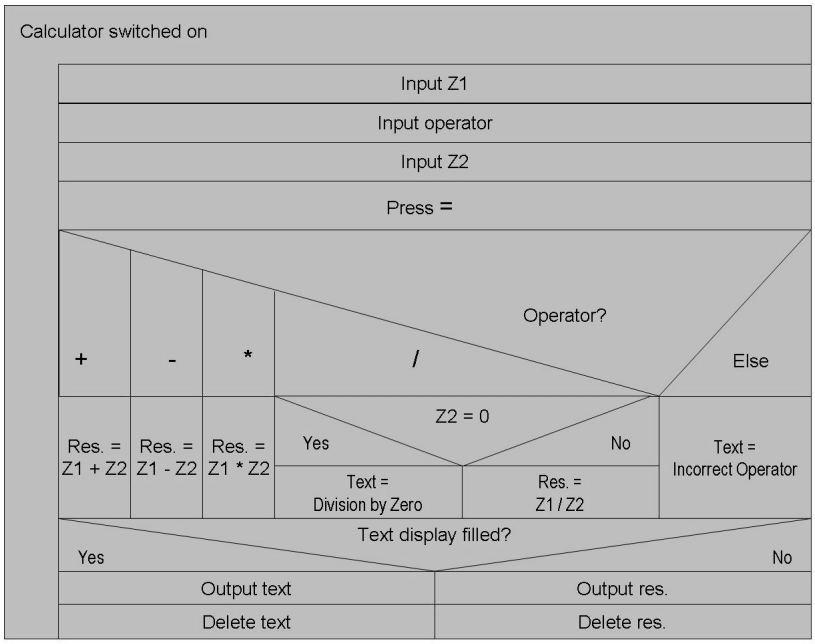


Figure 21: Structogram

2. If you have time:

   Create a structogram for the “making a call” requirement. Remember that there are cordless phones and phones with handsets. Some phones have functions to save numbers, while others do not. Decide whether you are to take a call or make a call yourself.

   a) The solution is to be discussed during the course.

Lesson Summary

You should now be able to:

- Implement solutions to problems in structograms

Internal Use SAP Partner Only

Unit Summary

You should now be able to:

- Identify problems
- Devise system-independent solutions to these problems
- Implement solutions to problems in program flowcharts (PFCs)
- Implement solutions to problems in structograms

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 2

Flow of an ABAP Program

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Explain the three levels (presentation, application, and database layers)
- Describe the process of communication between these levels
- Describe general ABAP syntax
- Use ABAP syntax
- Use transaction SE38 (ABAP Editor)
- Create local programs independently
- Describe the ABAP Editor functions
- Develop, activate, and test simple programs

Unit Contents

Lesson: System Architecture

Lesson Overview

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

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the three levels (presentation, application, and database layers)
- Describe the process of communication between these levels

Business Example

You work for the company "Calculate & Smile" and are to begin developing ABAP programs using SAP NetWeaver. To do this, you need to know how communication works across the three levels (presentation, application server, and database server).

System Architecture and ABAP Program

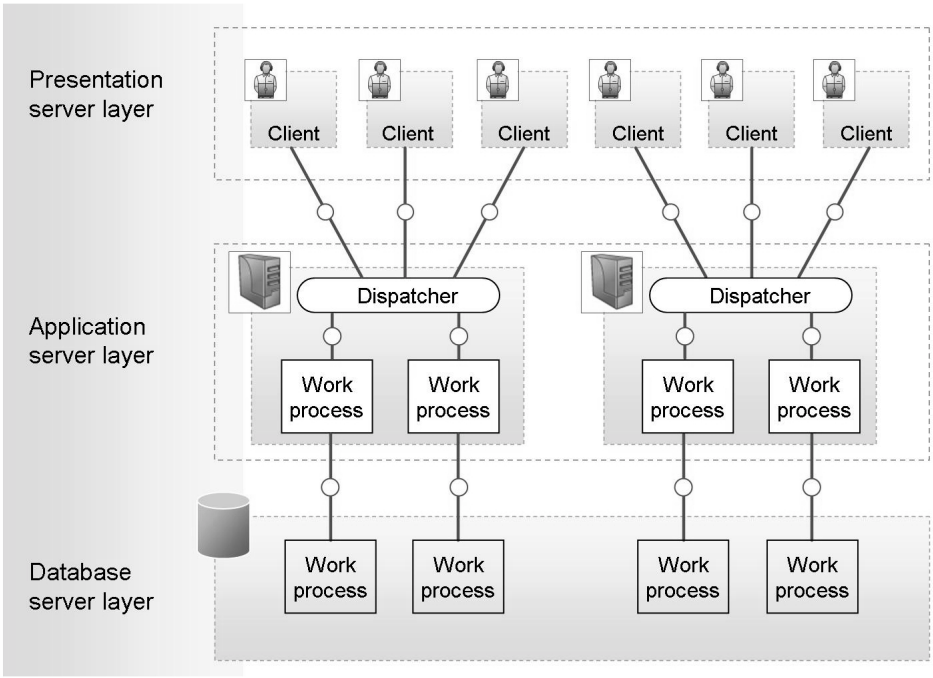


Figure 22: Client/Server Architecture of SAP NetWeaver Application Server

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

In *SAP NetWeaver Application Server*, presentations, application logic, and data storage can be assigned to different systems. This serves as the basis for the **scalability** of the system.

The lowest level is the **database level**. Here data is managed with the help of a relational database management system (RDBMS). This data includes, apart from application data, the programs and the metadata that the SAP system requires for self-management.

The ABAP programs run at the **application server level**, that is, both the applications provided by SAP and the ones you develop yourself. The ABAP programs read data from the database, process it, and store new data there if necessary.

The third level is the **presentation server level**. This level contains the user interface where each user can access the program, enter new data, and receive the results of a work process.

The technical distribution of software is independent of its physical location on the hardware. Vertically, all levels can be installed on top of each other on one computer or each level on a separate computer. Horizontally, the presentation and application servers can be divided among any number of computers. The horizontal distribution of database components, however, depends on the type of database installed.

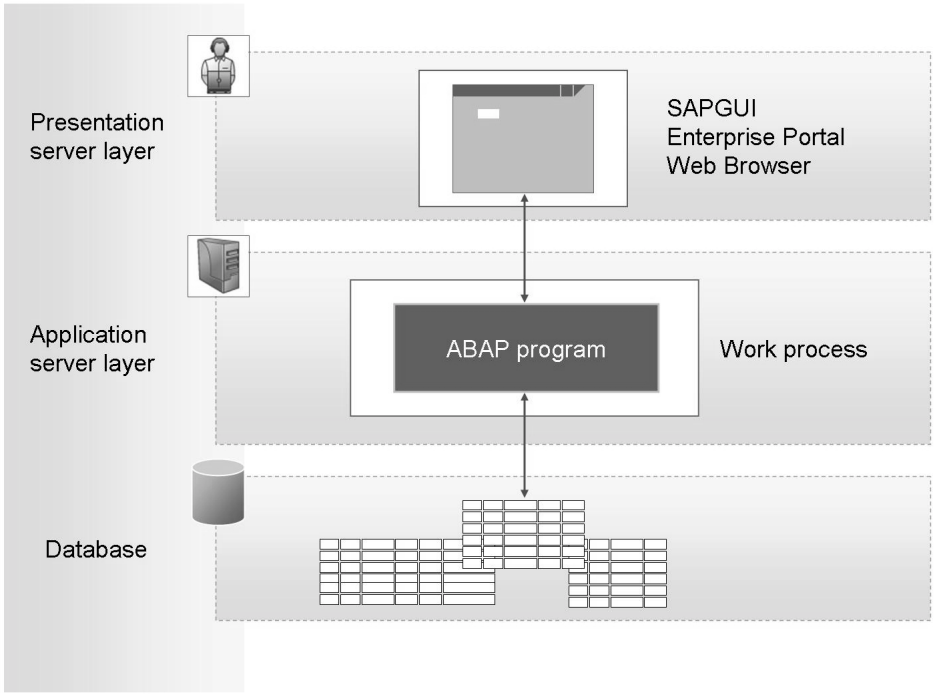


Figure 23: Excerpt for an ABAP Program

We can simplify this graphic for most of the topics that will be discussed during this course. The interaction between **one** user and **one** ABAP program will be of primary interest to us during this course.

The exact processes involved in user dispatching on an application server are secondary to understanding how to write an ABAP program. Therefore, we will be working with a simplified graphic that does not explicitly show the dispatcher and the work process. Certain slides will, however, be enhanced to include these details whenever they are relevant to ABAP programming.

ABAP programs are processed on the application server. The design of **user dialogs** and **database accesses** is of particular importance when writing application programs.

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

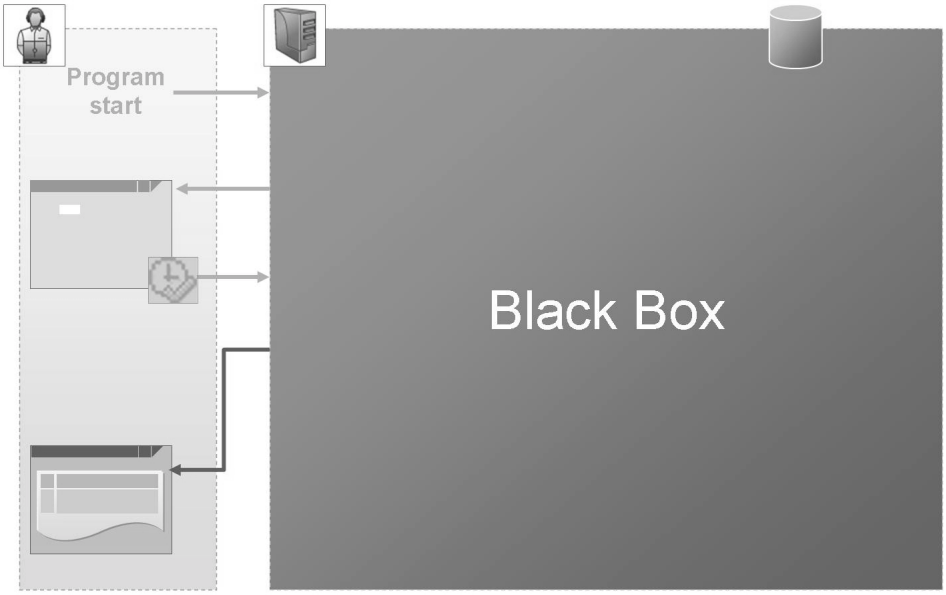


Figure 24: View for the User

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

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

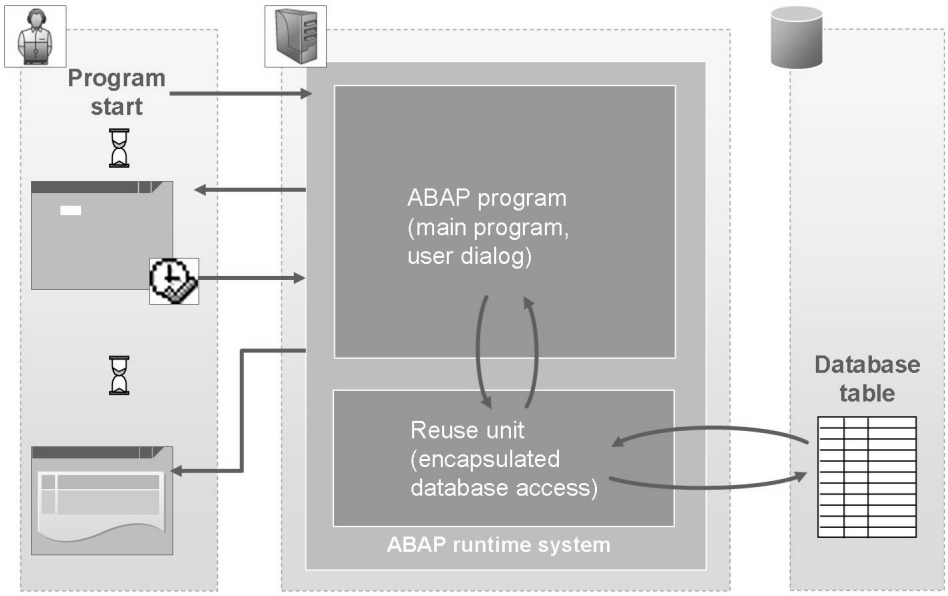


Figure 25: Interplay Between Server Levels and Program Flow

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

If a further user dialog is triggered from within the ABAP program, the system transmits the screen, and control is once again passed to the presentation server.

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

Process Flow of a Program with Selection Screen and List

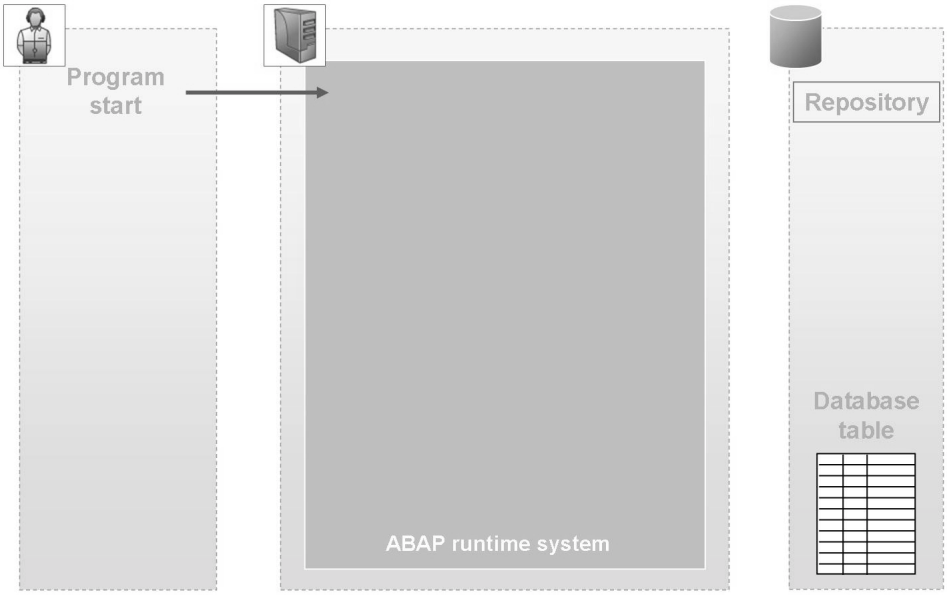


Figure 26: Program Start

Whenever a user logs on to the system, a screen is displayed. From this screen, the user can start an ABAP program through the menu path.

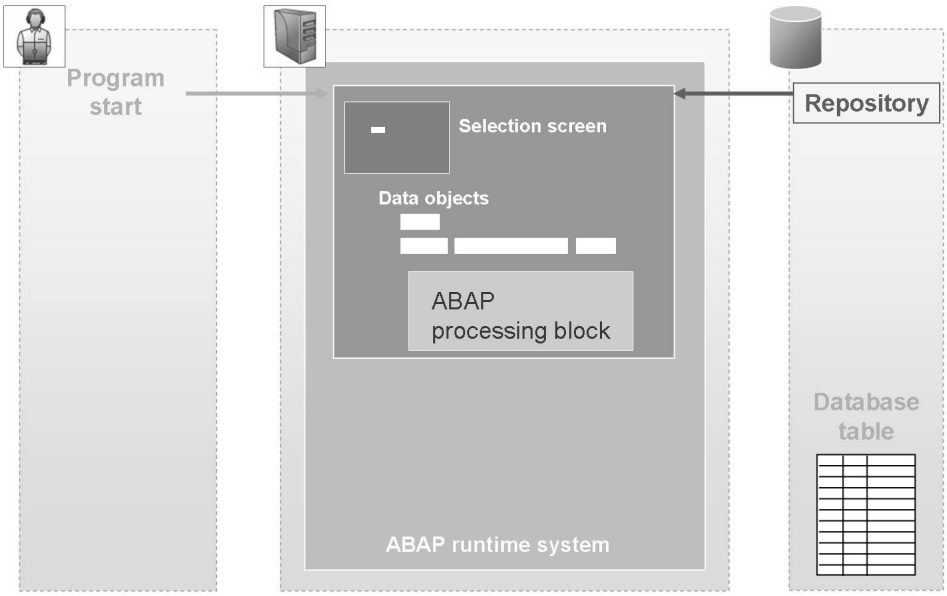


Figure 27: System Loads Program Context

In this case, the system first loads the program context onto the application server. The program context contains memory areas for variables and complex data objects, information on the screens for user dialogs, and ABAP processing blocks. The runtime system gets all this program information from the *Repository*, which is a special part of the database.

The sample program has a selection screen as the user dialog, a variable and a structure as data objects, and one ABAP processing block. The list used to display the data is created dynamically at runtime.

The ABAP runtime system controls the subsequent program flow.

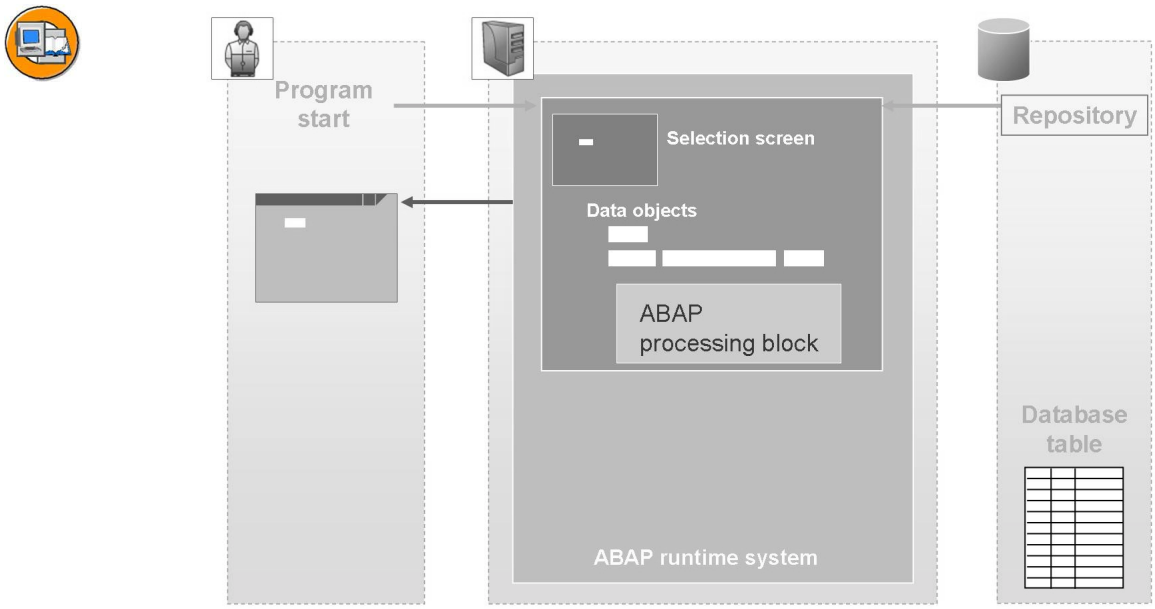


Figure 28: Runtime System Sends Selection Screen

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

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

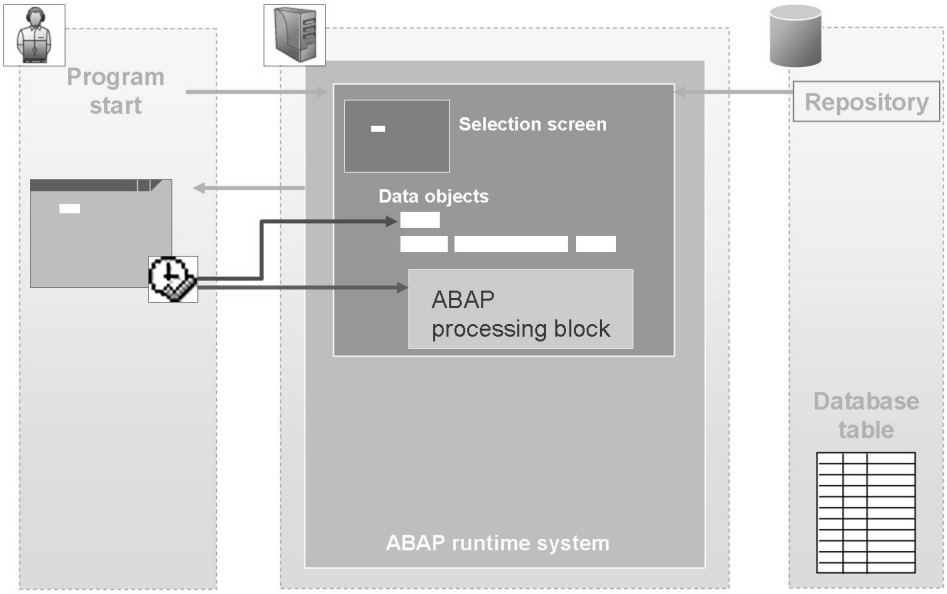


Figure 29: Input Values Are Inserted into Data Objects

As soon as the user has finished entering data on the selection screen, he or she can trigger further processing of the program by choosing *Execute*.

The entered data is automatically placed in its corresponding data objects in the program and the ABAP runtime system resumes control of processing.

In our simple program example, there is only one ABAP processing block. The ABAP runtime system triggers sequential processing of this ABAP processing block.

If the entries made by the user do not have the correct type, an error message is **automatically** triggered. The user must correct his/her entries.

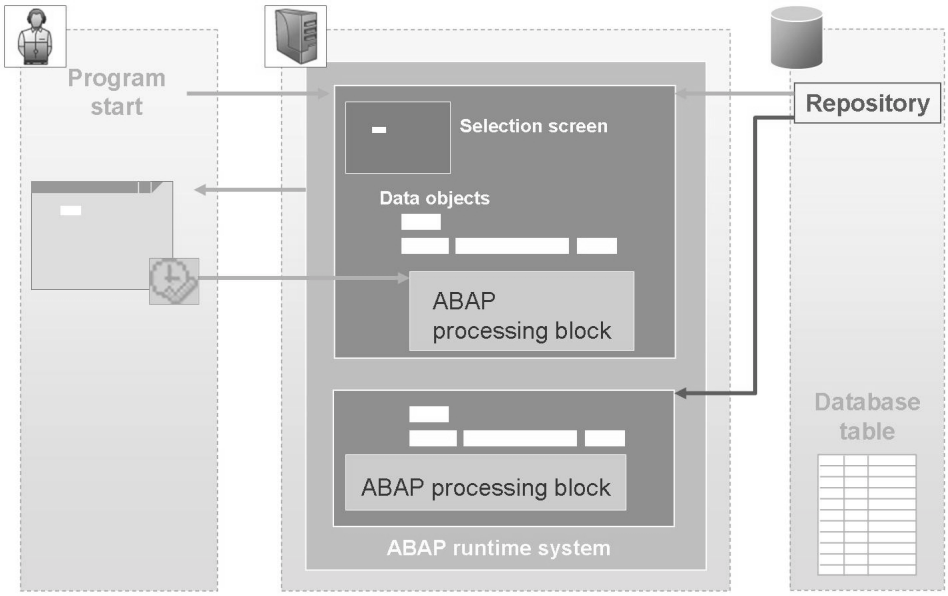


Figure 30: Reuse Unit Loading

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

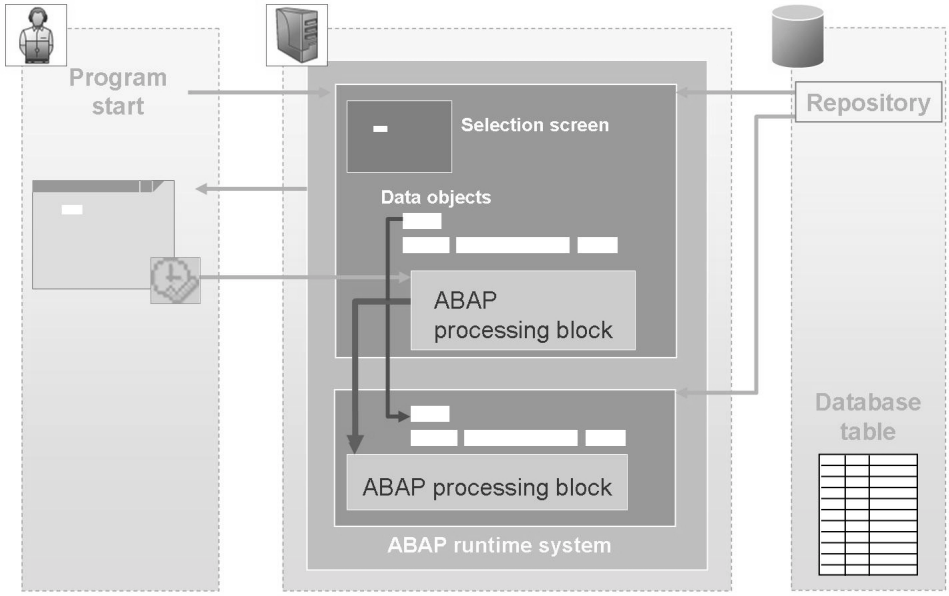


Figure 31: Reuse Unit Is Called

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

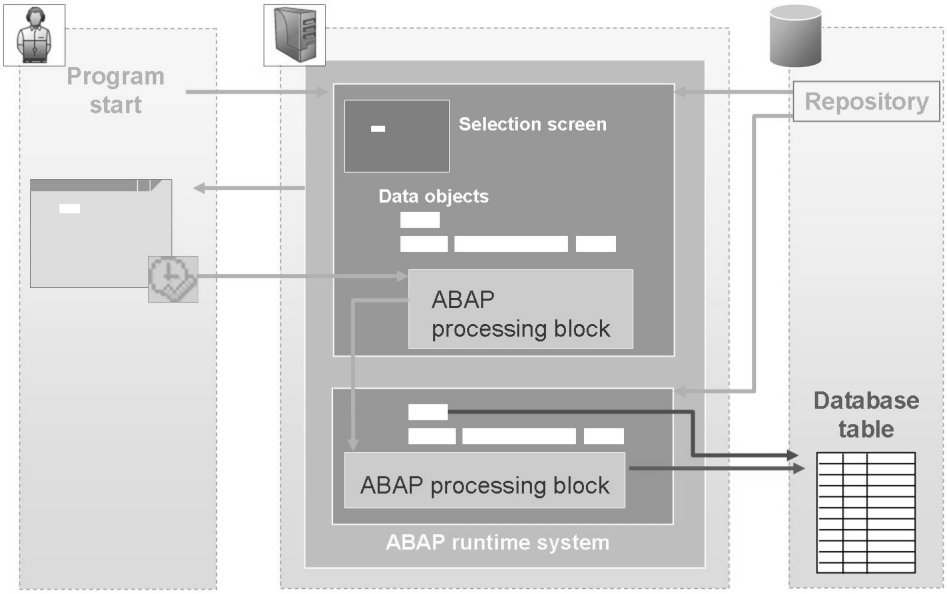


Figure 32: Program Requests Data Record from the Database

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

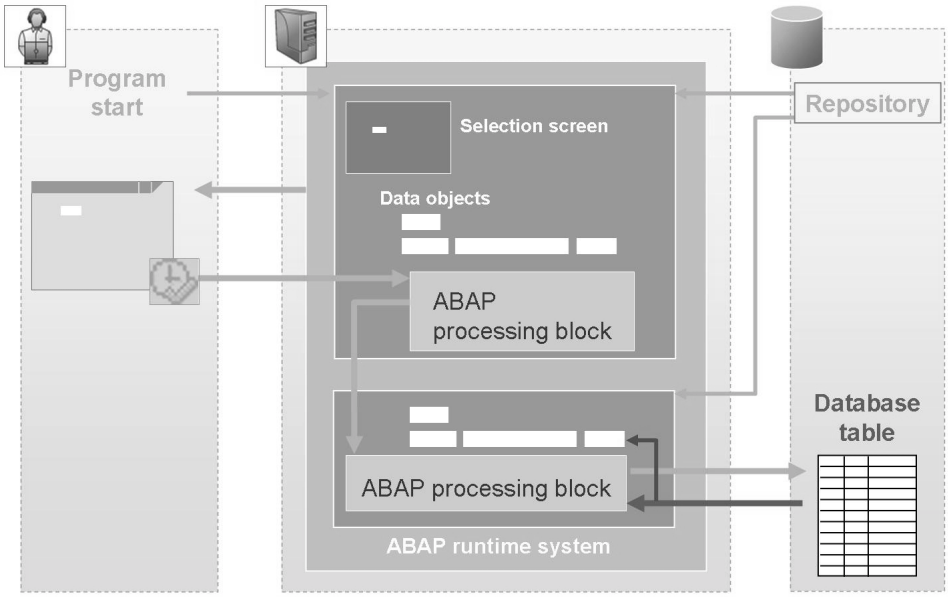


Figure 33: Database Supplies the Data Record

The database returns the requested data record to the program and the runtime system ensures that this data is placed in the appropriate data objects.

If a single record is accessed, this data object is usually a structure that contains relevant components for all the required database fields.

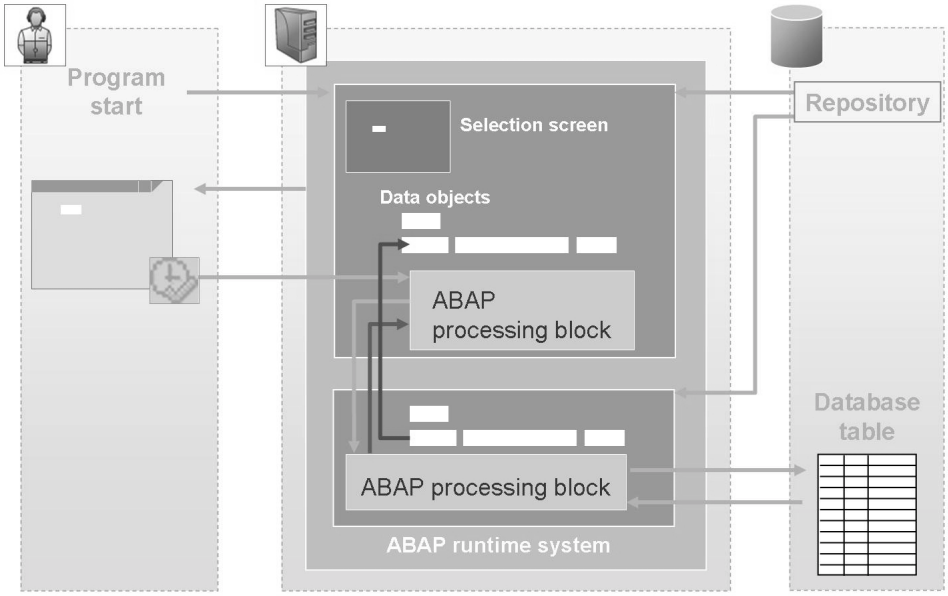


Figure 34: Reuse Unit Returns the Data

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

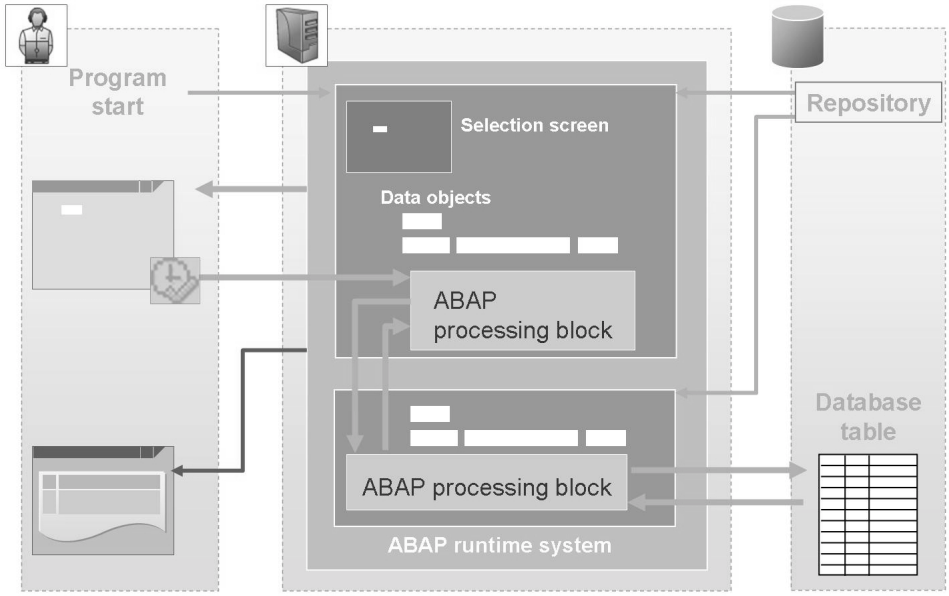


Figure 35: Runtime System Sends the List

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

Topic Overview

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

Unit 1

From the Problem to the Program - Introduction to the Structured Programming

Unit 2

ABAP Program Process

Unit 3

Introduction to ABAP Programming

Unit 4

Selection Screen, List, and Program Terminations

Unit 5

Example of Reuse Components in Subroutines

Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Explain the three levels (presentation, application, and database layers)
- Describe the process of communication between these levels

Internal Use SAP Partner Only

Lesson: General ABAP Syntax

Lesson Overview

This lesson outlines general ABAP syntax.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe general ABAP syntax
- Use ABAP syntax

Business Example

To be able to write ABAP programs at "Calculate & Smile", you need to know the general syntax of this programming language so that you can use it for development effectively.

Introduction to the ABAP Programming Language

The ABAP programming language ...

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

It is especially designed for dialog-based business applications.

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

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

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

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

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

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

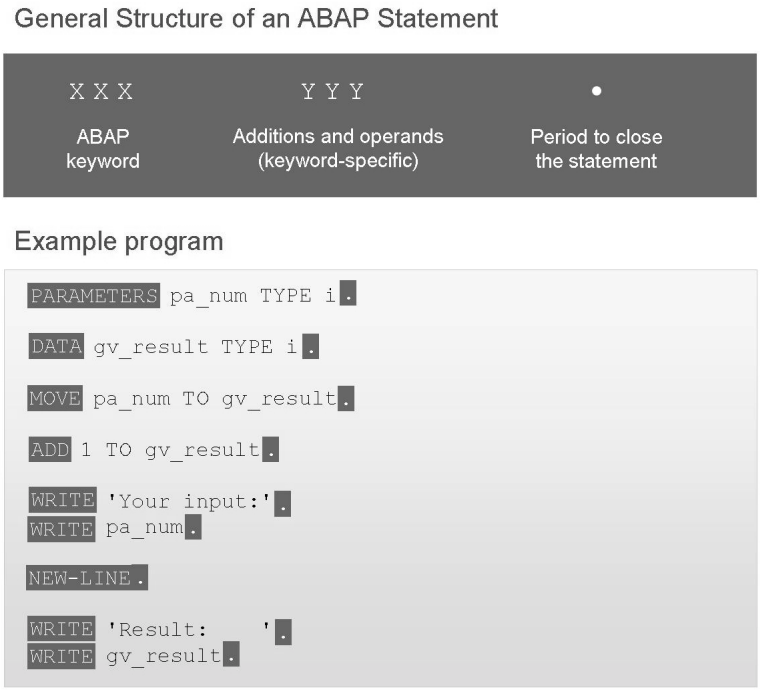


Figure 36: General ABAP Syntax I

For ABAP syntax, the following applies in general:

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

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

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

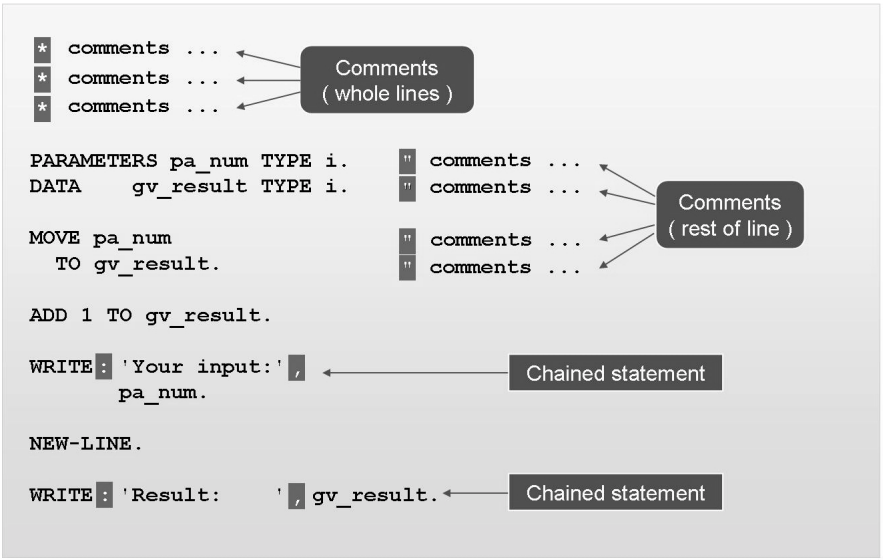


Figure 37: General ABAP Syntax II

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

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

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

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

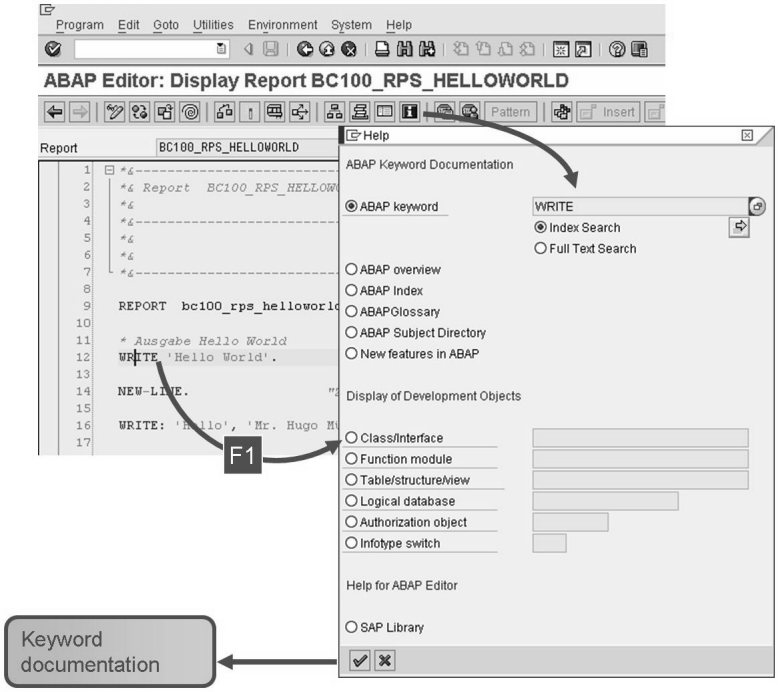


Figure 38: Keyword Documentation in the ABAP Editor

Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Describe general ABAP syntax
- Use ABAP syntax

Lesson: The Editor

Lesson Overview

This lesson provides an introduction to the functions of the new ABAP Editor, which was rolled out with SAP NetWeaver 7.0.

Lesson Objectives

After completing this lesson, you will be able to:

- Use transaction SE38 (ABAP Editor)
- Create local programs independently
- Describe the ABAP Editor functions
- Develop, activate, and test simple programs

Business Example

As an employee of "Calculate & Smile", you now want to familiarize yourself with the functions of the ABAP Editor. You must have knowledge of these functions to be able to develop and execute programs in ABAP.

Developing ABAP Programs

This section shows you how to create programs using the *ABAP Editor*. You will also learn about the purpose of program activation.

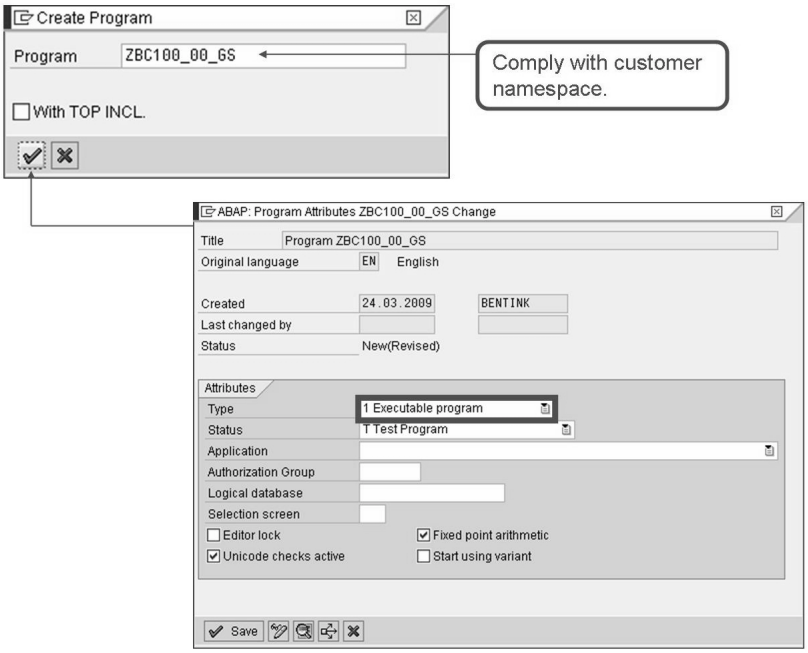


Figure 39: Creating an ABAP Program

Creating an ABAP Program

1. Navigate to the *ABAP Editor (SE38)*.

   Enter the name of the program you want to create (observe the customer namespace conventions). Confirm by choosing **`Create`**. If the program does not exist, the system goes to the dialog that lets you create a program.

2. Change the title to a self-explanatory short text and, within this course, always choose *Executable Program* as the program type. All other program attributes are optional. Refer to the **`F1`** Help for details.

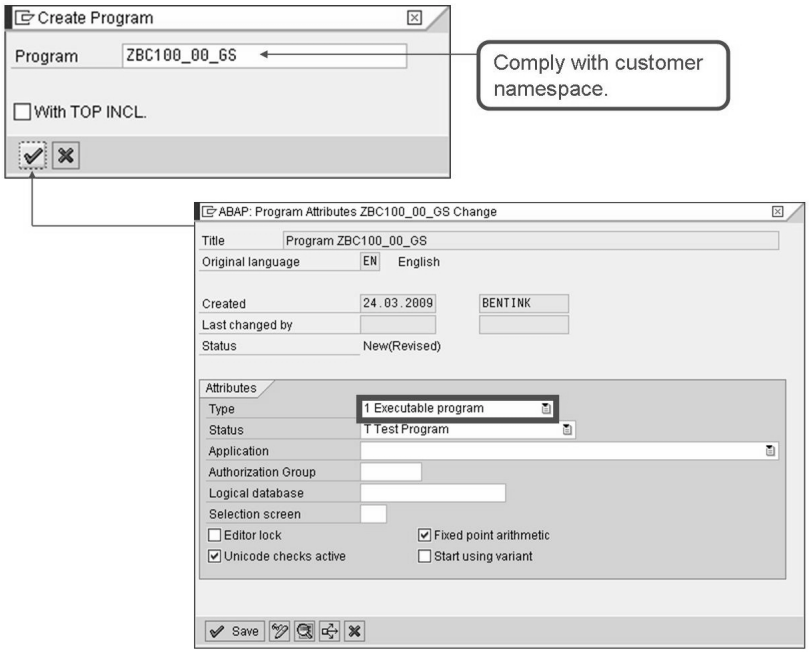


Figure 40: Creating an ABAP Program

The New Editor – General Features

- As of SAP Web Application Server 6.20, SP 59 SAP_BASIS
- As of SAP Web Application Server 6.40, SP 17 SAP_BASIS
- As of SAP NetWeaver 7.0
- SAP GUI 6.40 or higher required (at least SP10, but we recommend at least SP21)
- The relevant settings can be made in the Workbench settings:

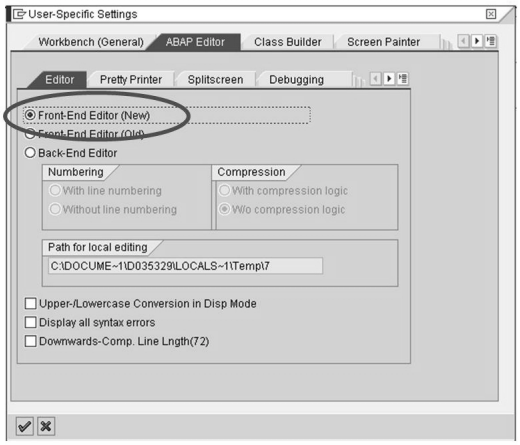


Figure 41: The New Editor – Availability

As of SAP NetWeaver 7.0, a new ABAP Editor is integrated in the ABAP Workbench. It is easier to use than the previous ones and also has interesting new functions. (The ABAP Editor is also available in a Support Package for SAP Web Application Server 6.20 or 6.40.) In addition to the correct version of the SAP system, you also require the relevant patch level for the front end (SAP GUI).

Developers can determine which version of the ABAP Editor they want to use for each individual development client: In transaction SE38, choose the menu path *Utilities* → *Settings*, *ABAP Editor* tab page.

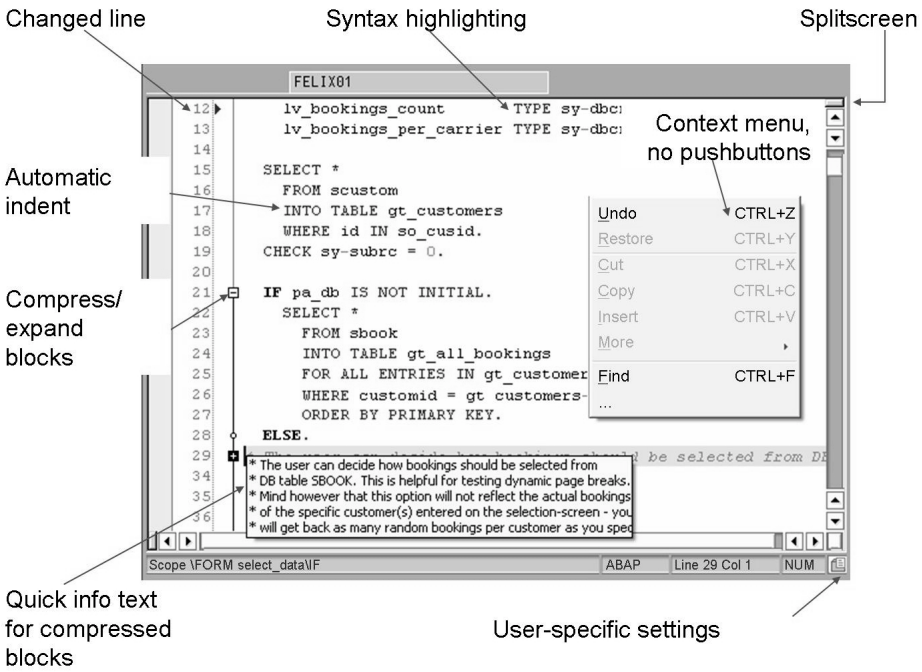


Figure 42: The New Editor – Features

The special features of the new editor include:

- Users can emphasize a large range of special syntax elements by applying specific formatting (font, font color, or size, as well as foreground and background color) to keywords, character strings, or the current row, for example.
- If the displayed lines of source code should extend beyond the current screen, you can split the editor into two parts (“splitscreen editor”) so that you can work in parallel on two different points in the program.
- If you have comment blocks or source code that are defined with a keyword at the beginning and at the end (for example, `IF...ENDIF` or `BEGIN OF <TYPE>... END OF <TYPE>.`), you can compress or expand these extracts to give yourself a better overview. To make these kind of extracts visible, but without showing them again, all you have to do is hold the cursor over the expand icon (a plus sign).

Note that you can no longer access familiar functions such as undo, search, or buffer using pushbuttons. You have to use the context menu (right mouse click) or a shortcut instead.

The Pretty Printer is still available to help you find your way through long and in particular nested source code by making appropriate indents. Additional orientation aids are available in the new editor. If you place the cursor inside a section of source code that has been defined with a keyword at the beginning and at the end (for example, `IF...ENDIF` or `BEGIN OF <TYPE>... END OF`

<TYPE>.), the expanded section of the relevant block is indicated by a line to the left of the source code. The area is also displayed at the bottom of the screen in the status bar. If you place the cursor between parentheses, the opening and closing parentheses are highlighted.

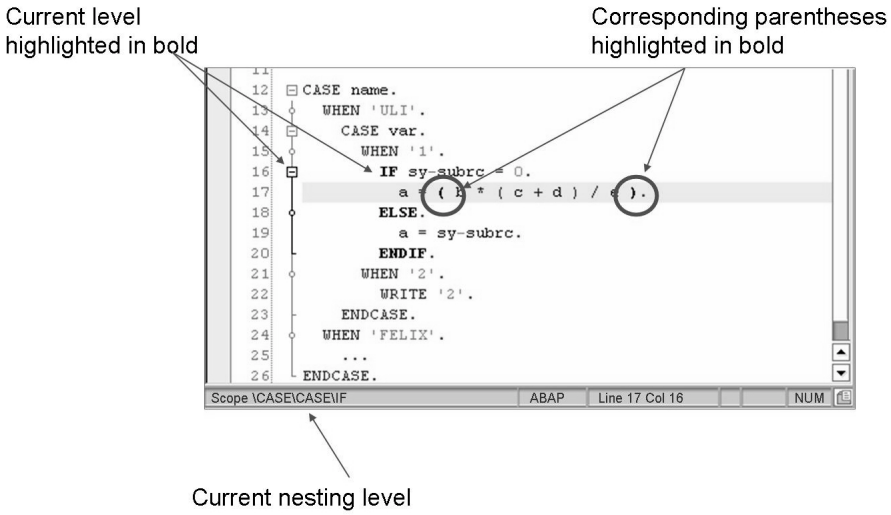


Figure 43: Identifying Blocks

In the classic line editor, you can select rectangular sections by choosing the key combination Ctrl + Y. In the new editor, you can do this by choosing Alt + left mouse button. You can copy a block that has been selected in such a way to the buffer or move it to the right with the tab key (or to the left by choosing Shift + tab).

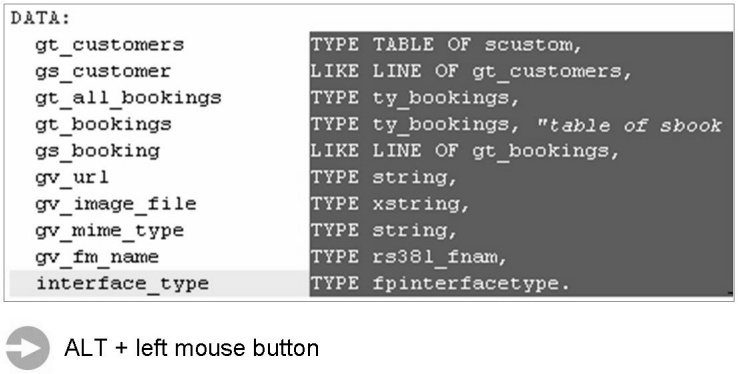


Figure 44: Block Selection

The status column is used to display breakpoints and bookmarks.

Code Hints and Code Templates

One of the main strengths of the new editor are code hints, which reduce the amount of typing to be done by the developer.

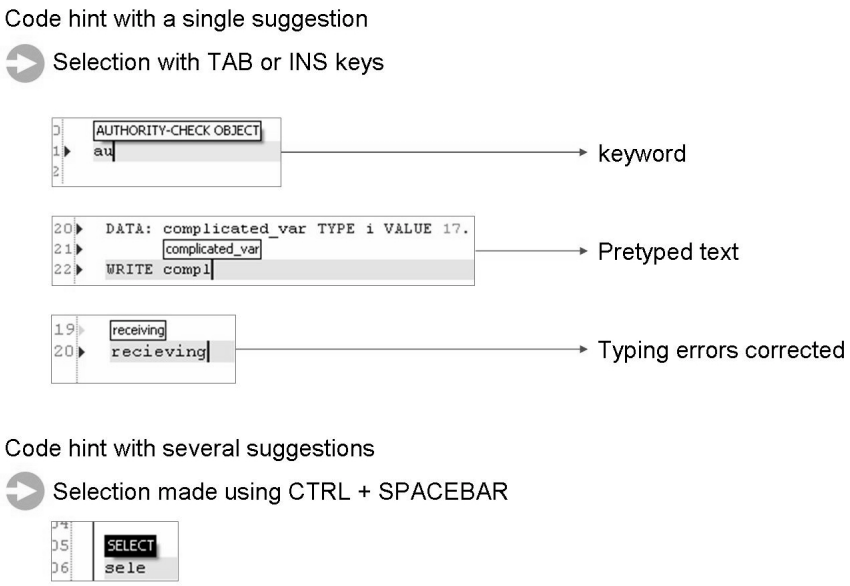


Figure 45: Code Hints

If the code hint is shown black on white, this means there is only **one** available suggestion. If the code hint is shown white on black, there are several available suggestions.

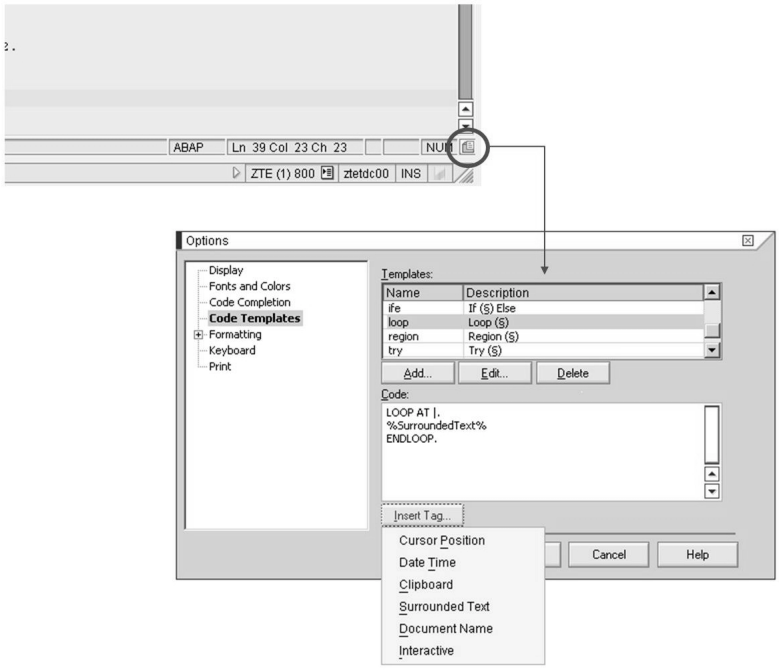


Figure 46: Code Templates

A range of code templates are already delivered with the editor that are proposed together with keywords, spelling suggestions, and pre-typed labels.

They are inserted using the tab or insert key, providing the proposal is still displayed as Quick Info (Tooltip). You can then copy the proposal by placing the cursor at the end of the term used to define the code template and choosing Ctrl + Enter.

To define your own code templates, click on the options icon in the lower right corner of the editor and choose *Code Templates*.

You can also insert predefined tags in code templates. For example, you can determine where the cursor should appear after you have inserted the template. You can also insert dynamic patterns: Any placeholder you enclose with percent signs, for example, %class name%.

Editor Options

You can access additional options such as font colors, shortcuts or indents, and the help function, using the Options icon in the lower right corner of the editor.

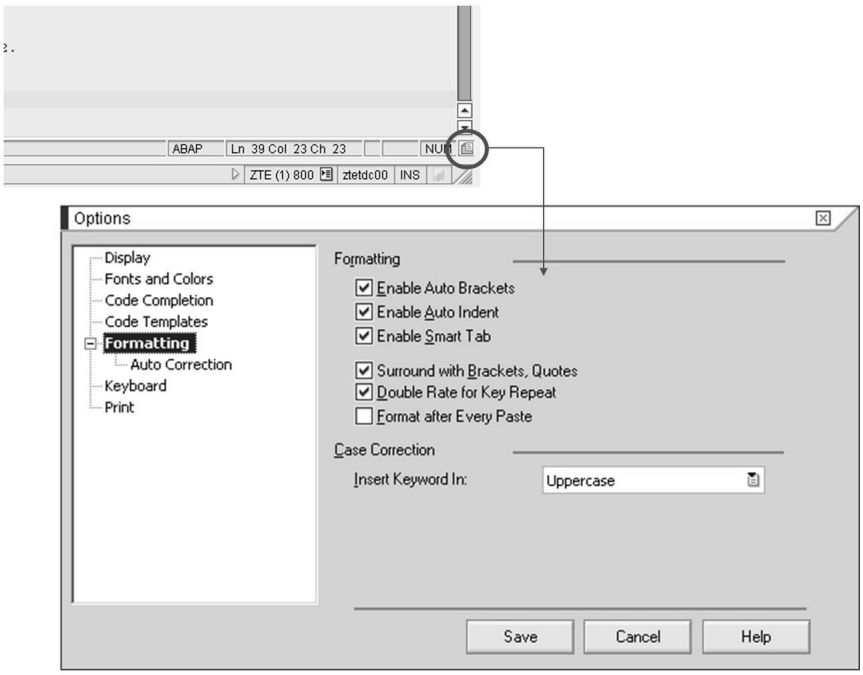
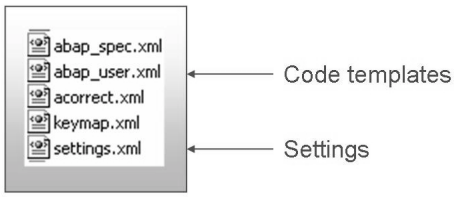


Figure 47: Editor Settings

All editor settings are delivered with initial values that are stored in special XML files. (With Microsoft Windows, they are normally stored in the directory C:\Program Files\SAP\FrontEnd\SAPgui\ab4_data.) As soon as the editor is used within a SAP GUI installation, these values are copied to a separate folder ab4_data located below the Local directory. You set this folder in the local layout of the SAP GUI.

Settings are saved as user-specific settings on the front-end computer as individual XML files:



<Directory_for_local_files>\ab4_data *

* → Options → Local Data → Directory for Local Data

Figure 48: Storing Editor Settings

Activating Programs

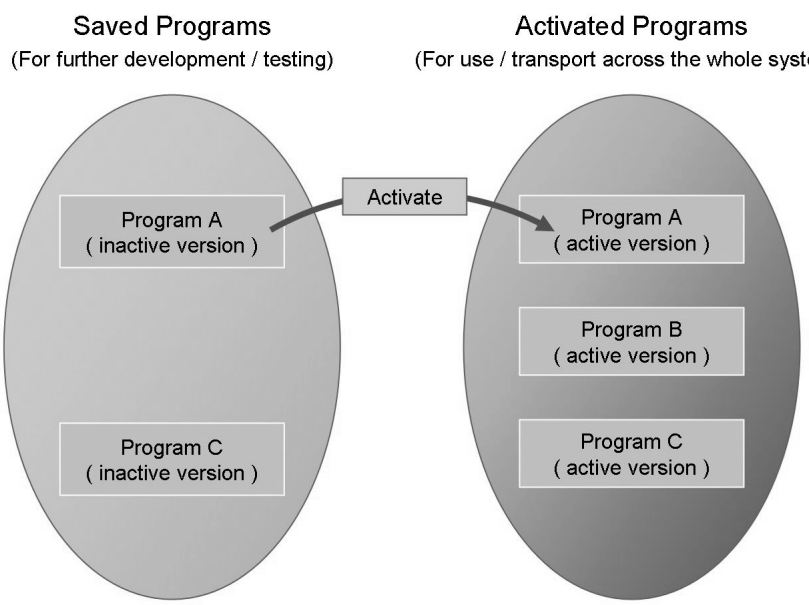


Figure 49: Inactive and Active Development Objects

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

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

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

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

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

The activation of an object includes the following functions:

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

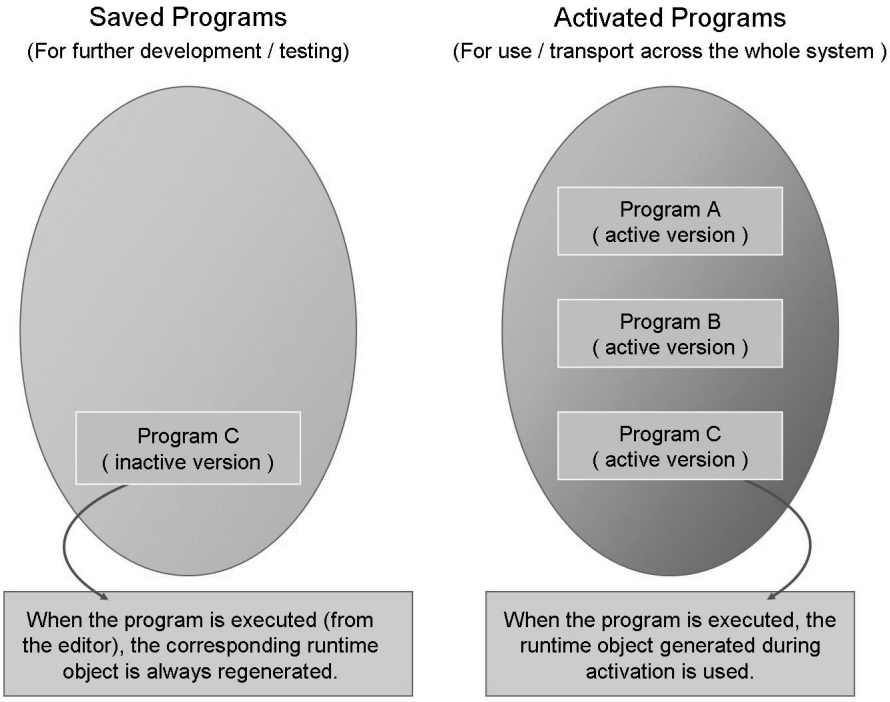


Figure 50: Generating Runtime Objects

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

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

- If you start your program using transaction SE38 (ABAP Editor), the active version is used. This means that the LOAD generated for the last activation is executed.
- By contrast, if you start the inactive version loaded to the editor using the *F8* button, a temporary runtime object is generated from it and executed.

In this way, it is possible to continue developing a program **without changing the current system status**. All changes to the program become “visible” system-wide only when the object is activated.

Exercise 4: Developing a Simple ABAP Program

Exercise Objectives

After completing this exercise, you will be able to:

- Create, edit, save, and process ABAP programs
- Check ABAP programs for correct syntax
- Test and activate ABAP programs

Business Example

As an employee of "Calculate & Smile", you are to familiarize yourself with the functions of the ABAP Editor. You must have knowledge of these functions to be able to develop and execute programs in ABAP.

Template:

None

Solution:

BC100_RPS_HELLOWORLD

Task 1:

Log on to the training system.

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

Task 2:

In your package, create an executable program named **ZBC100_##_HELLO**.

1. Create the program **ZBC100_##_HELLO**.
2. Make sure that the correct program type *Executable Program* is proposed in the next dialog box and set the status of the program to a meaningful value.

Task 3:

Edit the program. Output a text to the *list* and comment your code.

1. Output the text “Hello World” and comment your code. To do this, use the ABAP statement `WRITE`.

Continued on next page

2. Make sure that your name is output in a new line (statement `NEW-LINE`).
3. Now implement the output of the “Hello” text and your name with one **chained statement** instead of two separate statements.
4. Check your program for syntax errors. Test and activate it.

Internal Use SAP Partner Only

Solution 4: Developing a Simple ABAP Program

Task 1:

Log on to the training system.

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

   a) Carry out this step in the usual way.

2. Enter a new personal password.

   a) Carry out this step in the usual way.

Task 2:

In your package, create an executable program named **ZBC100_##_HELLO**.

1. Create the program **ZBC100_##_HELLO**.

   a) Carry out this step as described in the participants' handbook.

2. Make sure that the correct program type *Executable Program* is proposed in the next dialog box and set the status of the program to a meaningful value.

   a) Carry out this step as described in the participants' handbook. Set the program status to *Test Program*.

Task 3:

Edit the program. Output a text to the *list* and comment your code.

1. Output the text “Hello World” and comment your code. To do this, use the ABAP statement `WRITE`.

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

2. Make sure that your name is output in a new line (statement `NEW-LINE`).

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

3. Now implement the output of the “Hello” text and your name with one **chained statement** instead of two separate statements.

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

Continued on next page

4. Check your program for syntax errors. Test and activate it.

   a) Perform this step as described in the participants' handbook. Use the pushbuttons *Check (Ctrl+F2)*, *Direct (F8)*, and *Activate (Ctrl+F3)*.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_00_HELLOWORLD
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_00_helloworld.

* Hello world output
WRITE 'Hello World'.

NEW-LINE.                              "Line break

WRITE: 'Hello', 'Mr. Hugo Müller'.  "Display Mr. Hugo Müller
```

Lesson Summary

You should now be able to:

- Use transaction SE38 (ABAP Editor)
- Create local programs independently
- Describe the ABAP Editor functions
- Develop, activate, and test simple programs

Unit Summary

You should now be able to:

- Explain the three levels (presentation, application, and database layers)
- Describe the process of communication between these levels
- Describe general ABAP syntax
- Use ABAP syntax
- Use transaction SE38 (ABAP Editor)
- Create local programs independently
- Describe the ABAP Editor functions
- Develop, activate, and test simple programs

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 3

Introduction to ABAP Programming

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Distinguish between the elementary data objects and define and use data objects appropriately
- Describe the logical expressions and relational operators that are needed for branches and case distinction
- Use branches and case distinction correctly
- Explain the basic functions of the Debugger
- Debug simple programs
- Use loops effectively
- Explain the different ways to exit program units (such as loops) prematurely

Unit Contents

Lesson: Working with Elementary Data Objects, Assignments

Lesson Overview

This lesson explains the difference between data types and elementary data objects and shows you how to define and use these in a program. You will also learn some basic ABAP statements.

Lesson Objectives

After completing this lesson, you will be able to:

- Distinguish between the elementary data objects and define and use data objects appropriately

Business Example

You work for "Calculate & Smile" and need to familiarize yourself with data objects (variables) and data types. These are essential for the calculations. Data objects (variables) are defined and assignments made.

Data Types and Data Objects

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

Using Data Types

The following figure shows how data types can be used, as discussed in this course:

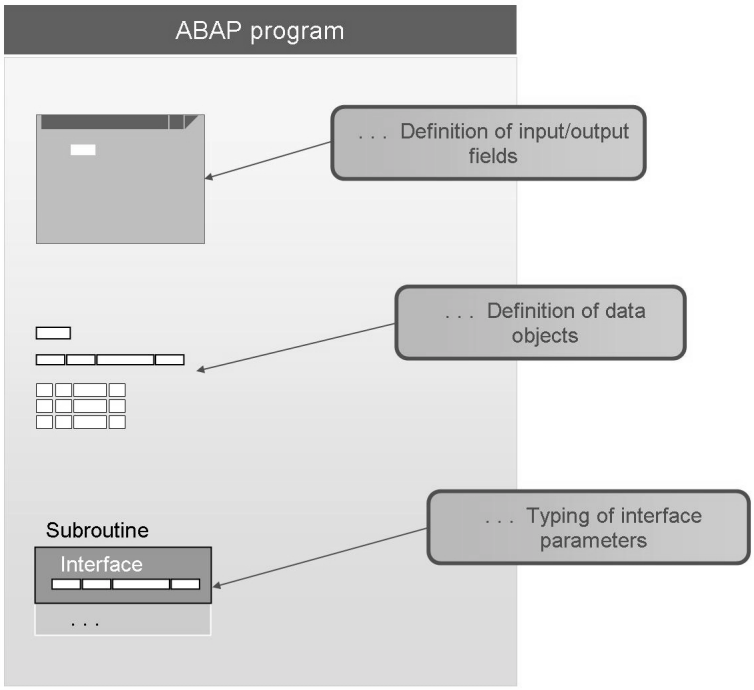


Figure 51: Using Data Types

Definition of data objects

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

Definition of interface parameters

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

Definition of input/output fields

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

This lesson focuses on how to use types for defining program-internal variables.

ABAP Standard Types

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

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

Complete ABAP Standard Types

D

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

T

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

I

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

F

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

STRING

Type for dynamic length character string

XSTRING

Type for dynamic length byte sequence (He**X**adecimal string)

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

Incomplete ABAP Standard Types

C

Type for character string (**C**haracter) for which the fixed length is to be specified

N

Type for numerical character string (**N**umerical character) for which the fixed length is to be specified

X

Type for byte sequence (He**X**adecimal string) for which the fixed length is to be specified

P

Type for packed number (**P**acked number) for which the fixed length is to be specified.

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

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

Local Data Types

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

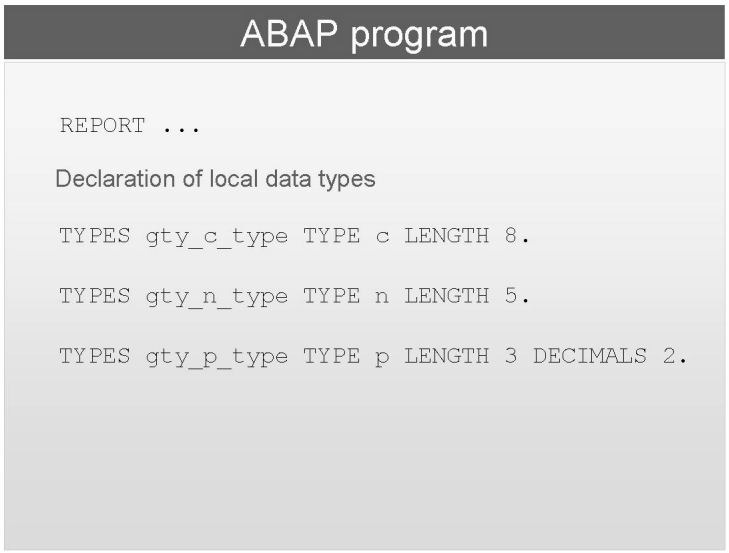


Figure 52: Declaration of Local Data Types

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

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

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

Global Data Types

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

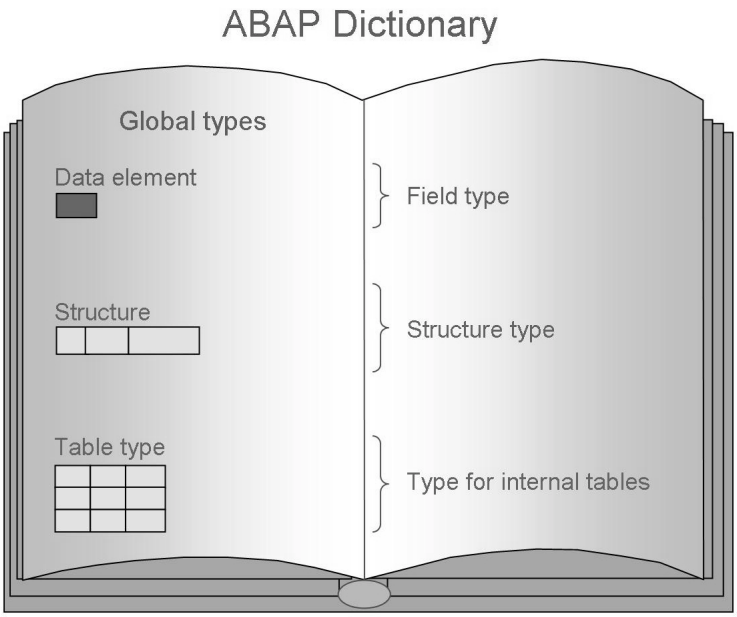


Figure 53: Global Data Types in the Dictionary

This lesson only focuses on the data elements and how to use them as field types. How other global data types are used is explained as part of the course BC400 (ABAP Workbench Foundations).

Definition of Variable Data Objects

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

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

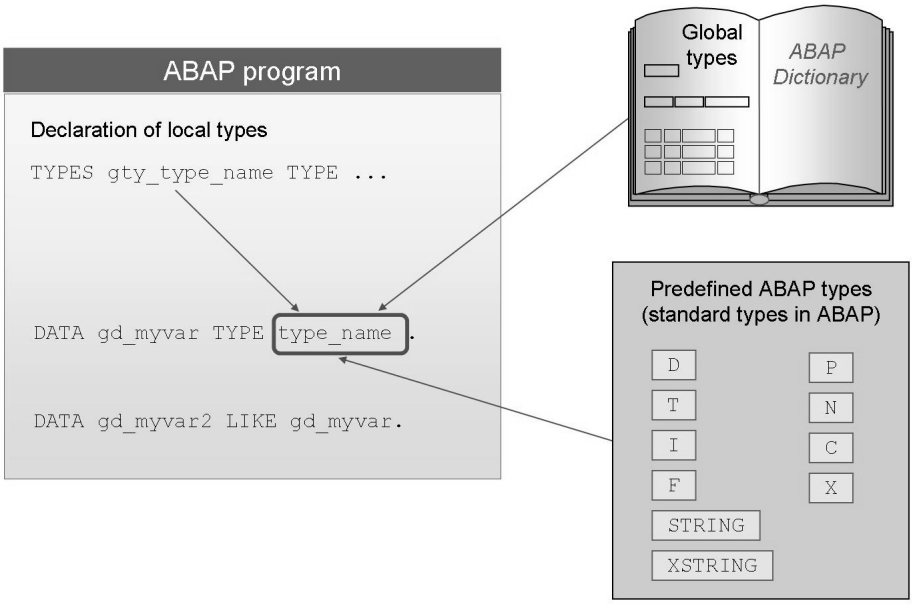


Figure 54: Defining Data Objects

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

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

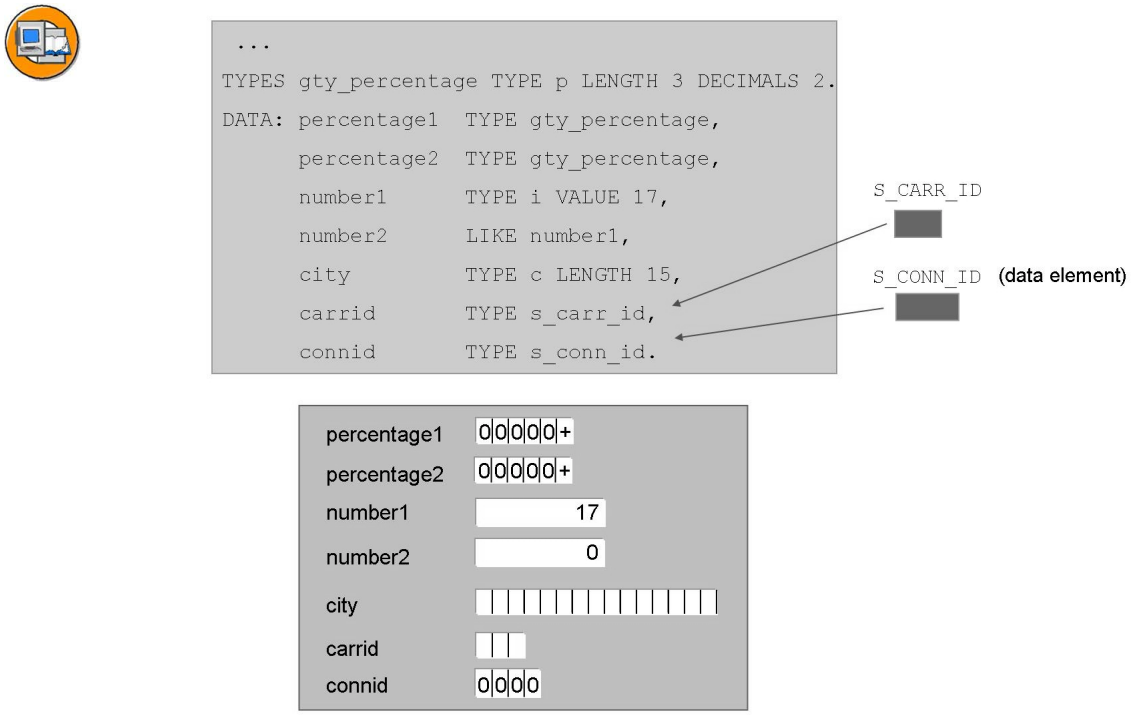


Figure 55: Examples of the Definition of Elementary Data Objects

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

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

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

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

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

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

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

Literals, Constants, and Text Symbols

Figure 56: Literals and Constants (Fixed Data Objects)

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

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

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

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

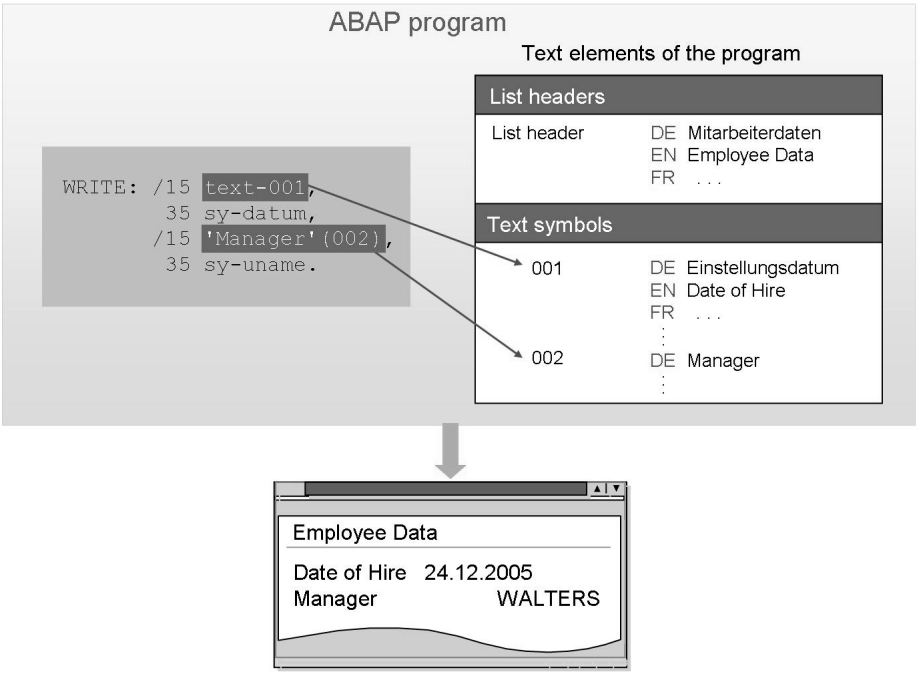


Figure 57: Text Symbols

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

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

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

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

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

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

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

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

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

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

Comparison: Local and Global Data Types

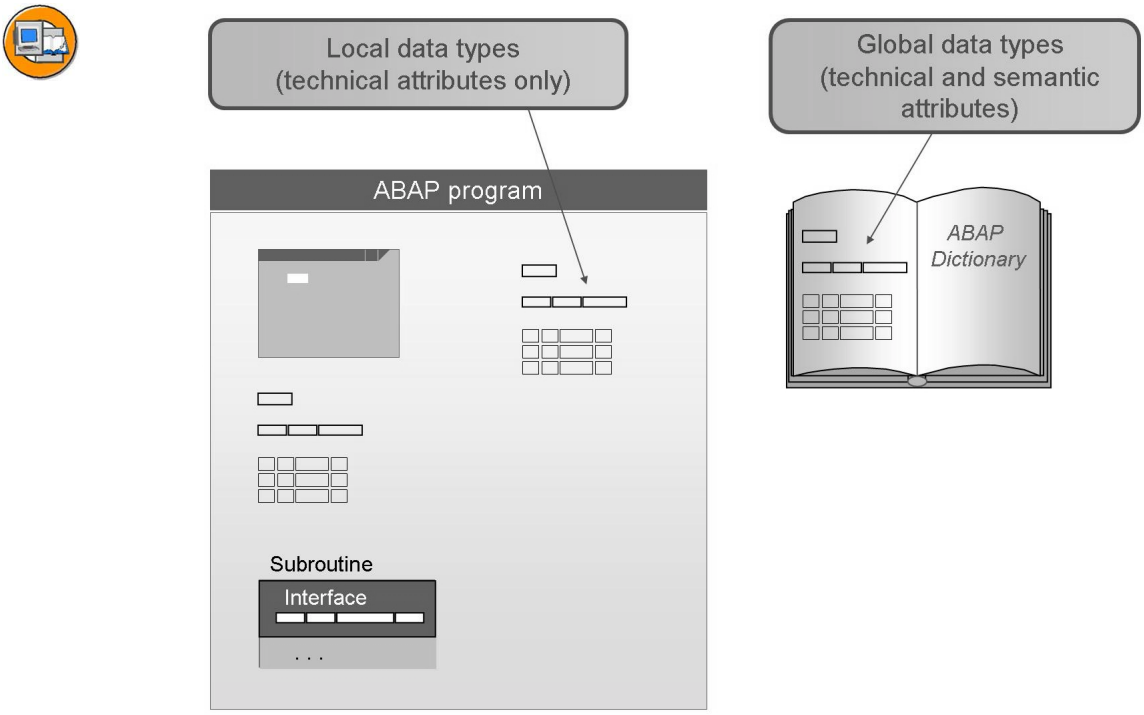


Figure 58: Local vs. Global Data Types

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

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

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

Basic ABAP Statements

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

Value Assignments

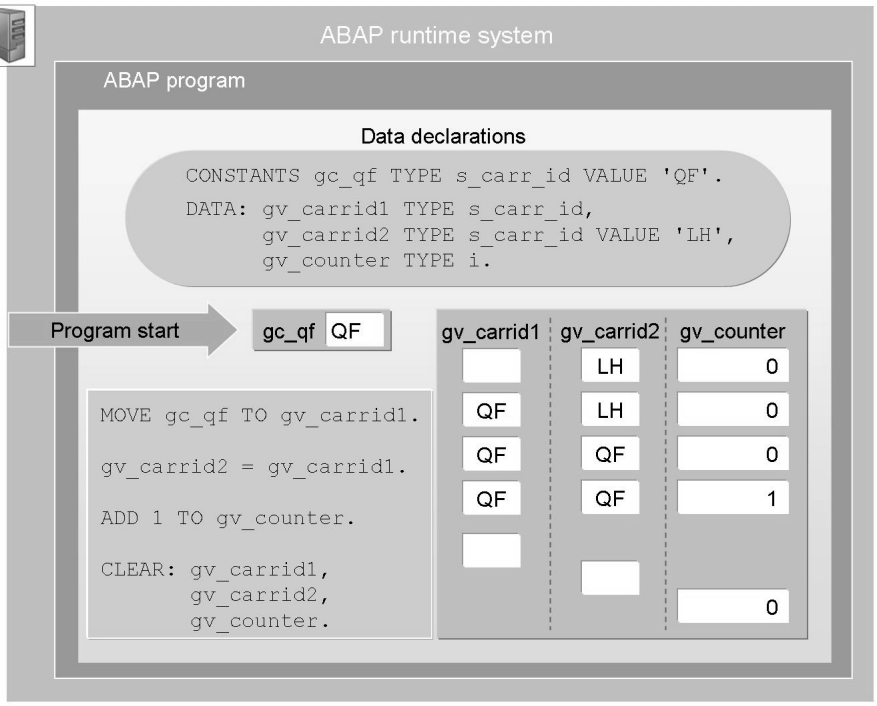


Figure 59: Value Assignments

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

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

- `MOVE gd_var1 TO gd_var2.`
- `gd_var2 = gd_var1.`

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

The `CLEAR` statement resets the content of a data object to the **type-related** initial value. For detailed information on the initial values for a particular type, refer to the keyword documentation for the `CLEAR` statement.

Calculations and Arithmetic Expressions

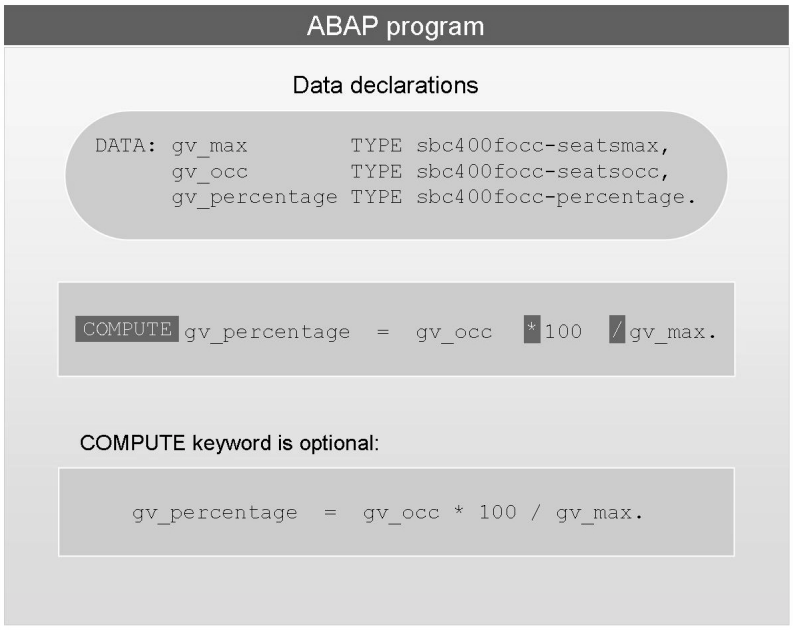


Figure 60: Calculations

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

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

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

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

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

In the case of functions, the opening parenthesis is part of the function name. The rest must again be separated by at least one space.

In general, the standard algebraic rules apply to the **processing sequence**: Expressions in parentheses come first, then functions, then powers, then multiplication/division, and finally addition/subtraction.

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

Internal Use SAP Partner Only

Exercise 5: Basic ABAP Statements

Exercise Objectives

After completing this exercise, you will be able to:

- Define elementary data objects
- Assign values
- Perform calculations

Business Example

You are to create a simple ABAP program for the four basic arithmetic operations. The values are to be assigned to the data objects (variables) directly in the program. The result is to be displayed in a list.

Template:

None

Solution:

BC100_DOS_COMPUTE

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_COMPUTE**.

Task 2:

Implement the basic arithmetic operations and output the result in a list.

1. Create an elementary data object (suggested name: **gv_erg**) using the DATA statement for the calculation result. Define it as a packed number type of 12 digits in length and 2 decimal places.
2. Use the DATA statement to create a data object for both the first and second operands (suggested names **gv_zahl1** and **gv_zahl2**). Define both as integers.
3. Assign the data object **gv_zahl1** the value 67, and the data object **gv_zahl2** the value 3.
4. Calculate the result once for each arithmetic operator.

Continued on next page

5. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

Task 3:

Activate and test your program.

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

Solution 5: Basic ABAP Statements

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_COMPUTE**.
   a) Perform this step in the same way as in the previous exercises.

Task 2:

Implement the basic arithmetic operations and output the result in a list.

1. Create an elementary data object (suggested name: **gv_erg**) using the DATA statement for the calculation result. Define it as a packed number type of 12 digits in length and 2 decimal places.
   a) See the source code excerpt from the model solution.
2. Use the DATA statement to create a data object for both the first and second operands (suggested names **gv_zahl1** and **gv_zahl2**). Define both as integers.
   a) See the source code excerpt from the model solution.
3. Assign the data object **gv_zahl1** the value 67, and the data object **gv_zahl2** the value 3.
   a) See the source code excerpt from the model solution.
4. Calculate the result once for each arithmetic operator.
   a) See the source code excerpt from the model solution.
5. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

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

Task 3:

Activate and test your program.

1. Activate your program.
   a) In the *ABAP Editor* toolbar, choose the pushbutton.
2. Test your program.

Continued on next page

a) In the *ABAP Editor* toolbar, choose the pushbutton.

b) Supply the selection screen parameters with values and choose .

c) Check the result. Test the message output in the event of an error too.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_DOS_COMPUTE
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_dos_compute.

DATA: gv_zahl1 TYPE i,
      gv_zahl2 TYPE i,
      gv_erg   TYPE p LENGTH 12 DECIMALS 2.

gv_zahl1 = 67.
gv_zahl2 = 3.
* Display operands
Write: / 'Number1', gv_zahl1, 'Number2:', 30 gv_zahl2.
ULINE.

*Addition
gv_erg = gv_zahl1 + gv_zahl2.
Write: / 'Result of Addition:', 30 gv_erg.
ULINE.

CLEAR gv_erg.
*Subtraction
gv_erg = gv_zahl1 - gv_zahl2.
Write: / 'Result of Addition:', 30 gv_erg.
ULINE.

CLEAR gv_erg.
*Multiplication
gv_erg = gv_zahl1 * gv_zahl2.
Write: / 'Result of Multiplication:', 30 gv_erg.
ULINE.

CLEAR gv_erg.
```

Continued on next page

```
*Division
gv_erg = gv_zahl1 / gv_zahl2.
Write: / 'Result of Divisioon:', 30 gv_erg.
ULINE.
```


Lesson Summary

You should now be able to:

- Distinguish between the elementary data objects and define and use data objects appropriately

Lesson: Logical Expressions and Relational Operators

Lesson Overview

This lesson explains how to use conditional branches and logical expressions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the logical expressions and relational operators that are needed for branches and case distinction

Business Example

As an employee of "Calculate & Smile", you need to learn about logical expressions and relational operators. Without these, branches and case distinctions cannot be realized.

Logical Expressions

The result of a logical expression **log_exp** is either true or false. Since ABAP does not have a Boolean data type for true or false values, logical expressions cannot be assigned to data objects. Instead, logical expressions are used for formulating conditions in control statements or other statements. Logical expressions can have relational operators or other language elements.

Boolean Operators and Using Parentheses

Operator	Meaning
AND	When multiple logical expressions **log_exp** are joined with **AND**, a new logical expression is formed that is true when all logical expressions **log_exp** are true. If one of the logical expressions is false, the operation is also false.
OR	When multiple logical expressions **log_exp** are joined with **OR**, a new logical expression is formed that is true if at least one of the logical expressions **log_exp** is true. Only if all of the logical expressions are false is the operation also false.
NOT	When a logical expression **log_exp** is negated with **NOT**, a new logical expression is formed that is false if the logical expression **log_exp** is true, and vice versa.

Figure 61: Logical Expressions

The Boolean operators **AND** and **OR** join multiple logical expressions, whereas the operator **NOT** negates a logical expression. Logical expressions can be defined explicitly using parentheses **( )**.

When multiple Boolean operators are combined, all logical expressions that are not defined explicitly by parentheses are put in parentheses by the system, based on the following hierarchy:

- All **NOT** Boolean operators are combined with the adjacent logical expression to the right into one logical expression.
- All logical expressions joined with **AND** are combined into one logical expression.
- All logical expressions joined with **OR** are combined into one logical expression.

The logical expressions within one level of parentheses are processed from left to right. If the value of one logical expression determines the total value of that level, the remaining logical expressions are no longer evaluated.

Relational Operators

In logical expressions with relational operators, two operands with one relational operator are joined into one relational expression. The logical expression compares the content of operand1 (based on the operator) with the content of operand2.

Relational Operators for All Data Types

Operator	Meaning
=, EQ	Equal: True if the content of operand1 is equal to the content of operand2.
<>, NE	Not Equal: True if the content of operand1 is not equal to the content of operand2.
<, LT	Less Than: True if the content of operand1 is less than the content of operand2.
>, GT	Greater Than: True if the content of operand1 is greater than the content of operand2.
<=, LE	Less Equal: True if the content of operand1 is less than or equal to the content of operand2.
>=, GE	Greater Equal: True if the content of operand1 is greater than or equal to the content of operand2.

Figure 62: Relational Operators

The way in which the content is compared depends on the data types of the operands.

Hint:

- We recommend that you use either the operators =, <>, <, >, <=, >= or the operators EQ, NE, LT, GT, LE, GE.
- Obsolete operators: ><, =<, =>

Lesson Summary

You should now be able to:

- Describe the logical expressions and relational operators that are needed for branches and case distinction

Lesson: Branches and Case Distinction

Lesson Overview

In the lesson, you learn how to use branches and case distinctions in a program.

Lesson Objectives

After completing this lesson, you will be able to:

- Use branches and case distinction correctly

Business Example

As an employee of "Calculate & Smile", you need to learn about branches and case distinction (multiple branches) and use these in a program. You are to use statements that are executed only under certain conditions (conditional statement). By using branches, you are able to execute a specific part of a program depending on a condition. You need to learn the importance of these components since they enable a program to respond to different inputs and states.

Conditional Branches and Logical Expressions

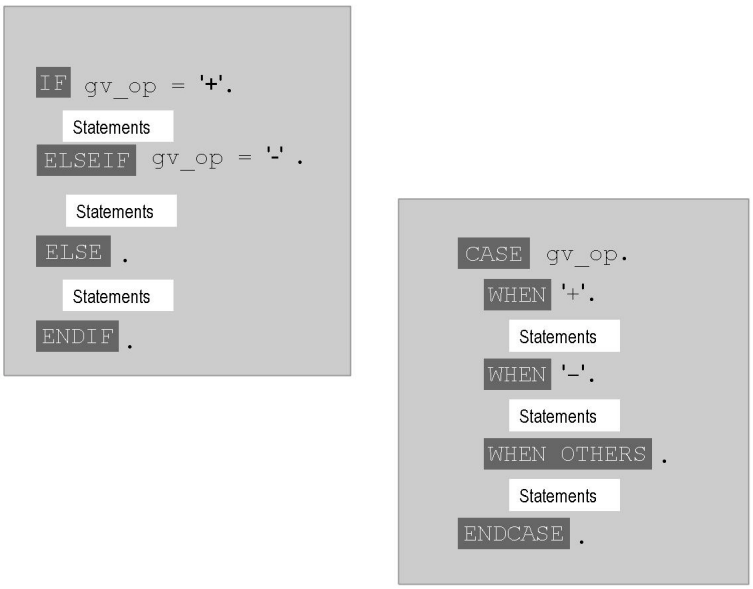


Figure 63: Conditional Branches

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

- In the **IF construct**, you can define **any logical expressions** as check conditions. If the condition is met, the system executes the relevant statement block. Otherwise, the condition specified in the ELSEIF branch (several are possible) is checked and so on. If none of the specified conditions are met, the system executes the ELSE branch, if it is available. ELSEIF and ELSE branches are optional. For detailed information about formulating a logical expression, refer to the keyword documentation on the IF statement.
- You can use the **CASE construct** to **clearly distinguish cases**. The content of the field specified in the CASE part is checked against the data objects listed in the WHEN branches to see whether they **match**. If the field contents match, the respective statement block is processed. If no comparison is successful, the system executes the OTHERS branch if it is available. Except for the first WHEN branch, all further additions are optional.

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

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

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

Negation

```
IF gv_op IS NOT INITIAL.
   Statements
ELSE.
   Statements
ENDIF.
```

AND and OR operations with parentheses

```
IF ( gv_op = '+' OR gv_op = '-'  or gv_op = '*' )
   AND gv_a > 0 and gv_b > 0.
   Statements
ELSEIF ( gv_op = '/' AND gv_b = 0.
   Statements
ENDIF.
```

Negation before logical conditions

```
IF NOT ( gv_op = '+' OR gv_op = '-' OR gv_op = '*'
   or gv_op = '/').
   Statements
ENDIF.
```

Figure 64: Examples: IF Statement

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

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

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

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

Exercise 6: Branches and Case Distinction

Exercise Objectives

After completing this exercise, you will be able to:

- Use branches and case distinction correctly

Business Example

You are to create a simple ABAP program for the four basic arithmetic operations. The result is to be displayed in a list.

Template:

None

Solution:

BC100_DOS_COMPUTE2

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_COMPUTE2**.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one arithmetic operator of type character with one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.
2. Calculate the result dependent on the specified arithmetic operator. Use the CASE statement for a case distinction.

Continued on next page

3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

Task 4:

Catch any errors that may occur.

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

Task 5:

Activate and test your program.

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

Solution 6: Branches and Case Distinction

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_COMPUTE2**.

   a) Perform this step in the same way as in the previous exercises.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one arithmetic operator of type character with one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

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

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.

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

2. Calculate the result dependent on the specified arithmetic operator. Use the CASE statement for a case distinction.

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

3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

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

Continued on next page

Task 4:

Catch any errors that may occur.

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

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

2. Display an error message on the list if the user tries to divide by zero.

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

Task 5:

Activate and test your program.

1. Activate your program.

   a) In the *ABAP Editor* toolbar, choose the pushbutton.

Continued on next page

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

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_DOS_COMPUTE2
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  BC100_DOS_COMPUTE2.

Data:   gv_int1  TYPE i,
        gv_int2  type i,
        gv_op    TYPE c,
        gv_result type p length 16 decimals 2.

gv_int1 = 21.
gv_int2 = 0.
gv_op   = '/'.

CASE gv_op.
  WHEN '+'.
    gv_result = gv_int1 + gv_int2.
  WHEN '-'.
    gv_result = gv_int1 - gv_int2.
  WHEN '*'.
    gv_result = gv_int1 * gv_int2.
  WHEN '/'.
    if gv_int2 = 0.
      write: / 'Division by zero!'.
    else.
      gv_result = gv_int1 / gv_int2.
    endif.
  When others.
    write: / 'Incorrect Operator!'.
ENDCASE.

write: / 'Result:', gv_result.
```

Lesson Summary

You should now be able to:

- Use branches and case distinction correctly

Internal Use SAP Partner Only

Lesson: Working with the Debugger

Lesson Overview

In this lesson, you learn how to execute programs in debugging mode. You also learn how to set breakpoints and change the content of data objects at runtime.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basic functions of the Debugger
- Debug simple programs

Business Example

As an employee of "Calculate & Smile", you need to use the Debugger to be able to track changes to data objects (variables) at runtime. You are also to be able to change the content of data objects at runtime yourself. The Debugger helps you better understand the program flow and is useful for finding errors.

Working with the *ABAP Debugger*

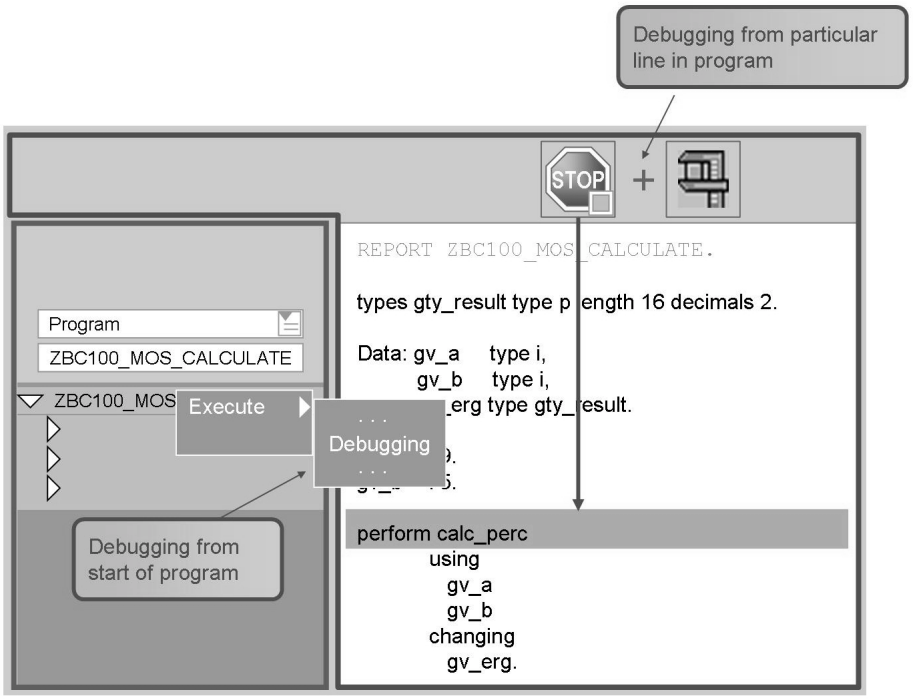


Figure 65: Execute a program in debugging mode

There are various ways to start a program in debugging mode from within the *ABAP Editor*:

a) After calling transaction SE38, select the program you require and choose the "Debugging" pushbutton or press *SHIFT + F5*.

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

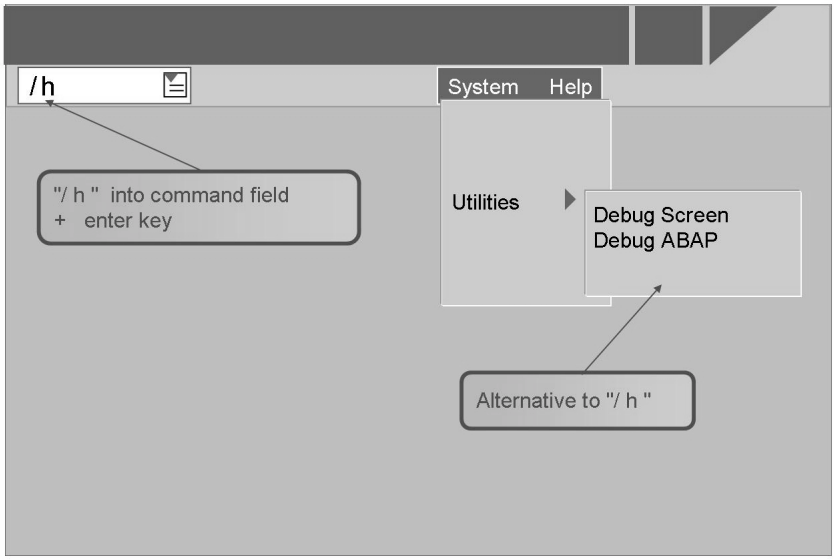
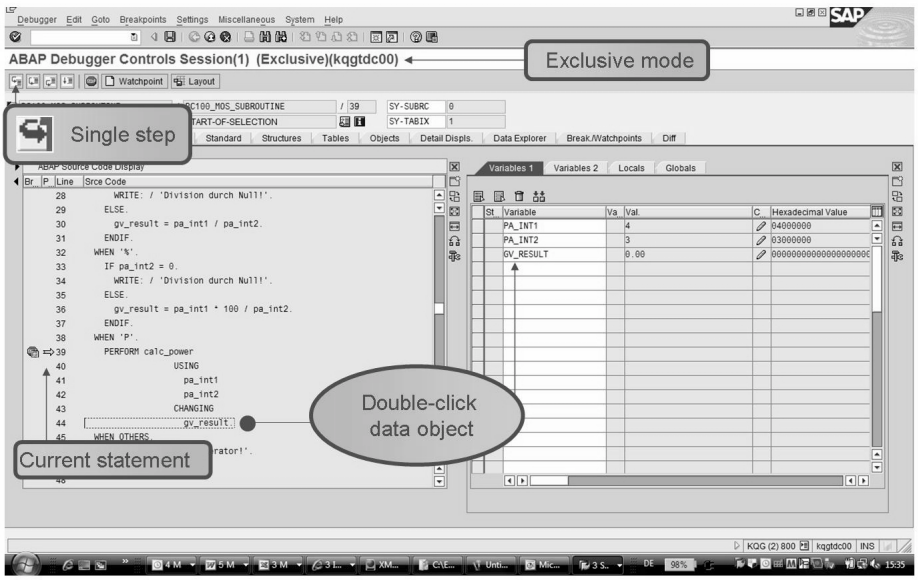


Figure 66: Switching to Debugging Mode at Runtime

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

a) Choose *System → Utilities → Debugging ABAP (or screen)*.

b) Enter **`/h`** in the command field in the standard toolbar and press **`Enter`**.



(Available as of SAP Web Application Server 6.40)

Figure 67: ABAP Debugger: Single Step and Field Contents

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

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

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

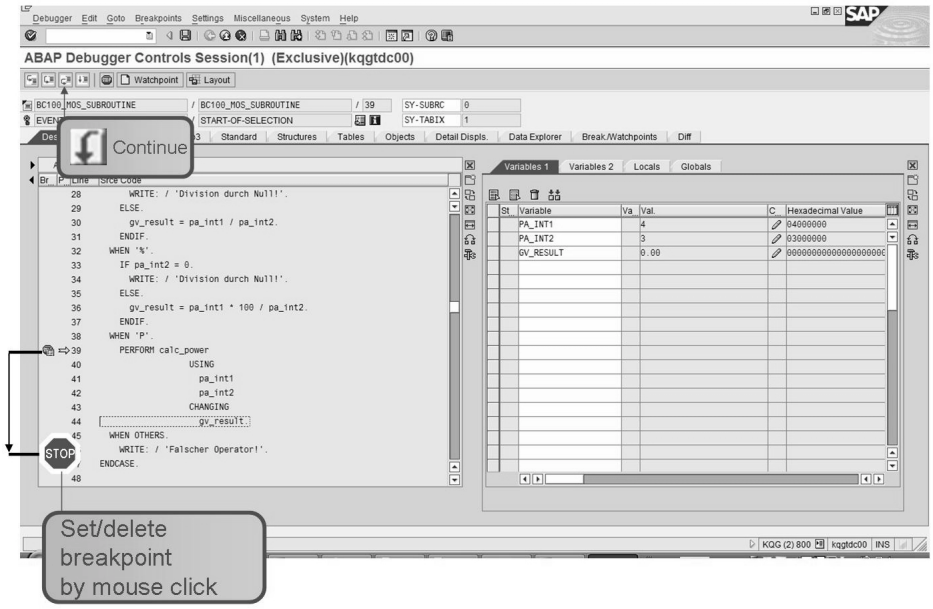


Figure 68: ABAP Debugger: Breakpoints

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

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

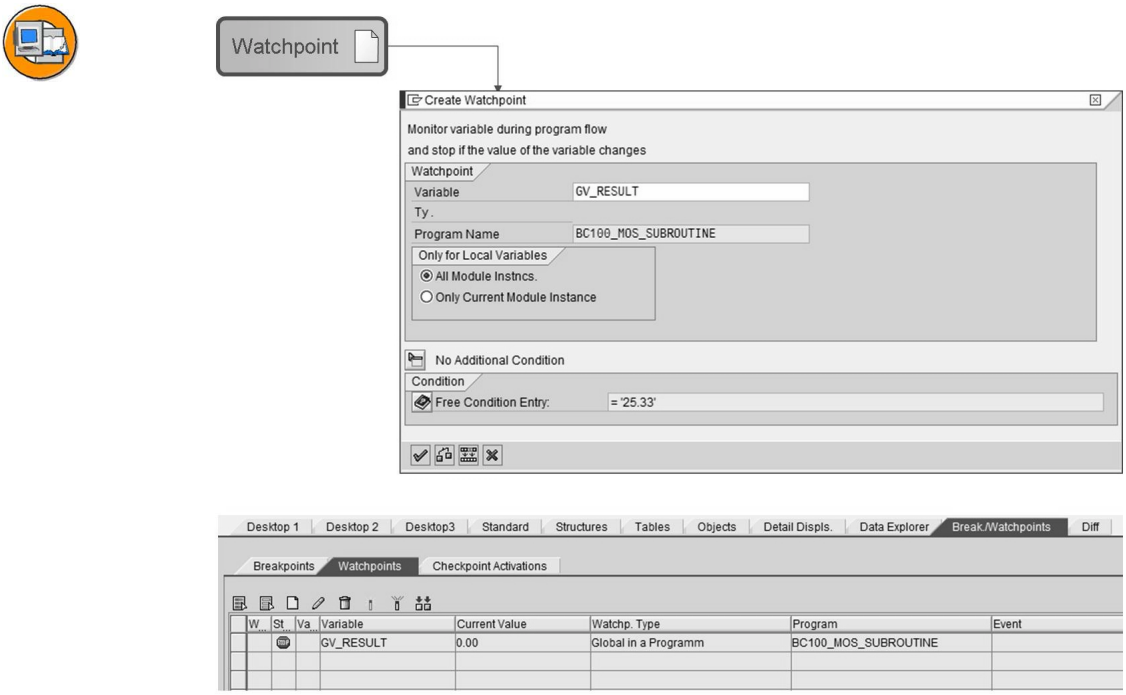


Figure 69: ABAP Debugger: Tracing Data Changes

Watchpoints are breakpoints that depend on the field content.

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

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

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

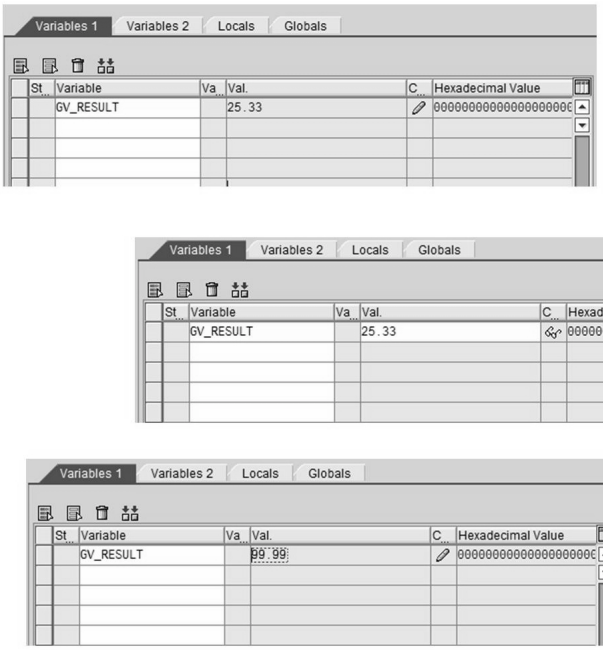


Figure 70: ABAP Debugger: Changing Field Contents

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

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

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

Exercise 7: Debugging Statements on Elementary Data Objects

Exercise Objectives

After completing this exercise, you will be able to:

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

Business Example

You are given a program and have to use the Debugger to work out its semantic behavior. This program currently returns no result and still contains errors. You are to change the program so that the errors are eliminated and the program functions as a calculator.

Template:

BC100_DET_DEBUG_EXERCISE

Solution:

BC100_DES_DEBUG_EXERCISE

Task 1:

Copy the program **BC100_DET_DEBUG_EXERCISE** (suggested name: **ZBC100_##_DEBUG_EXERCISE**).

1. Carry out this step in the usual manner.

Task 2:

Execute the program **ZBC100_##_DEBUG_EXERCISE** in debugging mode and trace the calculations.

1. Start the program BC100_##_DEBUG_EXERCISE in debugging mode.
2. First check the content of the data objects in the program at the start. To do this, include all the data objects defined in the program and the system field SY-INDEX in the variable view. Take a look at the typing of the fields too.

Continued on next page

Task 3:

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

1. Follow the program flow in single steps until you have run through the `WHILE` loop twice. In each case, track how the field contents change.
2. Set a breakpoint at the statement that assigns a new value to the data object GV_RES. Choose *Continue* to restart the program and observe the changes to the field contents. Repeat the last step until the program has run its course.
3. Execute the program again in debugging mode. Now analyze the change to the data object GV_RES by creating a watchpoint for the field. Choose *Continue* to restart the program and observe the change to the content of the data object and the system field SY-INDEX.

Task 4:

Output the result

1. Make sure that the result is output to list.

Task 5:

If you have time: Use the Debugger options to change the content of the data object **PA_INT2** in such a way that all statements in the program are executed. Make sure that the program does not terminate when dividing by zero.

1. If the operator is / or **%**, the content of the data object **PA_INT2** must not be zero.

Task 6:

If you have time: Eliminate the existing errors.

1. Eliminate the existing errors so that the calculator functions can be used. Use the Debugger to find the errors.

Solution 7: Debugging Statements on Elementary Data Objects

Task 1:

Copy the program **BC100_DET_DEBUG_EXERCISE** (suggested name: **ZBC100_##_DEBUG_EXERCISE**).

1. Carry out this step in the usual manner.

   a) Copy the program as normal.

Task 2:

Execute the program **ZBC100_##_DEBUG_EXERCISE** in debugging mode and trace the calculations.

1. Start the program BC100_##_DEBUG_EXERCISE in debugging mode.

   a) Display the program in the ABAP Editor (*SE38*) and then choose *Program → Test → Debugging*.

   b) Alternatively, in the editor, you can also set a breakpoint at the first executable statement and choose *F8*.

2. First check the content of the data objects in the program at the start. To do this, include all the data objects defined in the program and the system field SY-INDEX in the variable view. Take a look at the typing of the fields too.

   a) In the source code of the Debugger, double-click each of the following fields: PA_INT1, PA_INT2, PA_OP, GV_RES, and SY-INDEX. Scroll to the right of the variable display in order to display the field types.

Task 3:

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

1. Follow the program flow in single steps until you have run through the `WHILE` loop twice. In each case, track how the field contents change.

   a) You can carry out the single step analysis by choosing *Debugger → Single Step*, the  pushbutton, or the *F5* key.

Continued on next page

2. Set a breakpoint at the statement that assigns a new value to the data object GV_RES. Choose *Continue* to restart the program and observe the changes to the field contents. Repeat the last step until the program has run its course.

   a) Click once in the first column in front of the value assignment to set a breakpoint there.

   b) Choose *Debugger* → *Continue* from the menu, the pushbutton, or F8.

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

   a) To access the Debugger, repeat the first step in this exercise.

   b) In the source code of the Debugger, position the cursor on the data object GV_RES. Choose the *Watchpoint* pushbutton and confirm the following dialog box by choosing .

   c) To restart the program, choose *Continue*.

Task 4:

Output the result

1. Make sure that the result is output to list.

   a) See source code.

Continued on next page

Task 5:

If you have time: Use the Debugger options to change the content of the data object **PA_INT2** in such a way that all statements in the program are executed. Make sure that the program does not terminate when dividing by zero.

1. If the operator is / or **%**, the content of the data object **PA_INT2** must not be zero.

   a) Set a breakpoint before the calculation and choose *Continue* to run the program up to this point.

   b) Include the field PA_INT2 in the variable view. Double-click the icon in the variable view. Change the field value from 0 to any number and confirm the change by choosing **Enter**.

   c) Restart the program by choosing either *Single Step* or *Continue*.

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

Task 6:

If you have time: Eliminate the existing errors.

1. Eliminate the existing errors so that the calculator functions can be used. Use the Debugger to find the errors.

   a) See source text.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_DES_DEBUG_EXERCISE
*&
*&---------------------------------------------------------------------*
*& Exercise: BREAKPOINT at statement, watchpoint,
*&           change field value during debugging
*&---------------------------------------------------------------------*
REPORT  bc100_des_debug_exercise.


Parameters: pa_int1 type i default 2,
            pa_op   type c default 'P',
            pa_int2 type i default 8.

DATA: gv_res type p length 16 decimals 2.
```

Continued on next page


```
if ( pa_op = '/' or pa_op = '%' ) and pa_int2 = 0.
  write: / 'Division by zero!'.
else.
  case pa_op.
    when '+'.
      gv_res = pa_int1 + pa_int2.
    when '-'.
      gv_res = pa_int1 - pa_int2.
    when '*'.
      gv_res = pa_int1 * pa_int2.
    when '/'.
      gv_res = pa_int1 / pa_int2.
    when '%'.
      gv_res = pa_int1 * 100 / pa_int2.
    when 'P'.
      if pa_int2 = 0.
        gv_res = 1.
      else.
        gv_res = pa_int1.

        WHILE sy-index < pa_int2.
          gv_res =  gv_res * pa_int1.
        ENDWHILE.
      endif.
    when others.

  endcase.

  write: / 'Result:', gv_res.
endif.
```


Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Explain the basic functions of the Debugger
- Debug simple programs

Lesson: Loops

Lesson Overview

This lesson explains the different loop constructs and how they are used in a program.

Lesson Objectives

After completing this lesson, you will be able to:

- Use loops effectively

Business Example

As an employee of "Calculate & Smile", you have to develop the basic functions for division and multiplication yourself. To do this, you use program loops. Program loops enable certain parts of the program code to be repeated for as long as the loop condition is valid or until a termination condition is triggered.

Loops

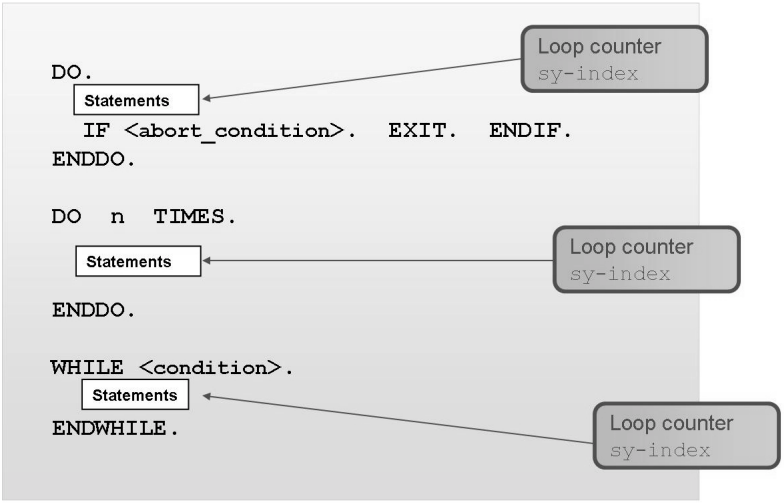


Figure 71: Loops

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

Unconditional/index-controlled loops

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

Precondition loops

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

Read loops

You can use the `SELECT` loop to read several entries of a database table in succession. In an internal table (table variable in the program), the same read function is implemented with the `LOOP` loop. Both read loops are covered in the course BC400 (ABAP Workbench Foundations) and are not part of this course.

System Fields

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

System field	Meaning
sy-mandt	Logon client
sy-uname	Logon name of user
sy-langu	Logon language of user
sy-datum	Local date of ABAP system
sy-uzeit	Local time of ABAP system
sy-tcode	Current transaction code
sy-repid	Name of current ABAP program
sy-index	Loop counter for DO and WHILE loops
...	...

Figure 72: Some Interesting System Fields

You will find a complete list of system fields in the keyword documentation under the term “System Fields”.

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

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

```
REPORT ...

DATA text TYPE string VALUE 'One Two Three'.

WHILE sy-subrc = 0.
  REPLACE space WITH '-' INTO text.
ENDWHILE.
```

Figure 73: Return Code of an ABAP Statement

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

Dialog Messages

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

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

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

Figure 74: Dialog Messages

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

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

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

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

Exercise 8: Using Loops

Exercise Objectives

After completing this exercise, you will be able to:

- Use the DO and WHILE loops in programs

Business Example

You are to use the “multiplication” and “division” functions by applying loop logic.

Template:

None

Solution:

BC100_REP_LOOP

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_LOOP**.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one as arithmetic operator of type “C” of one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.
2. Calculate the result based on the specified arithmetic operator. Use the “Do” loop for multiplication and (if you have time) the “While” loop for division. In the case of division, the whole number is to be calculated. Use the CASE statement for the case distinction.

Continued on next page

3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

Task 4:

Catch any errors that may occur.

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

Task 5:

Activate and test your program.

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

Solution 8: Using Loops

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_LOOP**.
   a) Perform this step in the same way as in the previous exercises.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one as arithmetic operator of type "C" of one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.
   a) See the source code excerpt from the model solution.

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.
   a) See the source code excerpt from the model solution.
2. Calculate the result based on the specified arithmetic operator. Use the "Do" loop for multiplication and (if you have time) the "While" loop for division. In the case of division, the whole number is to be calculated. Use the CASE statement for the case distinction.
   a) See the source code excerpt from the model solution.
3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

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

Continued on next page

Task 4:

Catch any errors that may occur.

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

Task 5:

Activate and test your program.

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

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_REP_LOOP
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  BC100_REP_LOOP.

Data:   gv_int1  TYPE i,
        gv_int2  type i,
        gv_op    TYPE c,
        gv_result type p length 16 decimals 2.

gv_int1 = 1326.
gv_int2 = 3.
gv_op   = '/'.
```

Continued on next page

Internal Use SAP Partner Only

```
CASE gv_op.
  WHEN '+'.
    gv_result = gv_int1 + gv_int2.
  WHEN '-'.
    gv_result = gv_int1 - gv_int2.
  WHEN '*'.
    if gv_int1 >= gv_int2.
      do gv_int2 times.
        gv_result = gv_result + gv_int1.
      enddo.
    else.
      do gv_int1 times.
        gv_result = gv_result + gv_int2.
      enddo.
    endif.
  WHEN '/'.
    if gv_int2 = 0.
     WRITE: / 'Division by zero!'.
    else.
      while gv_int1 > 0.
        gv_int1   = gv_int1 - gv_int2.
        gv_result = sy-index.
      endwhile.
    endif.
  When others.
    WRITE:  / 'Incorrect operator!'.
ENDCASE.

write: / 'Result:', gv_result.
```


Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Use loops effectively

Lesson: Exiting Program Units

Lesson Overview

This lesson continues by presenting the options for exiting a program unit.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the different ways to exit program units (such as loops) prematurely

Business Example

As an employee of "Calculate & Smile", you want to exit a program loop prematurely because a state has been reached that makes it necessary to exit the loop (for example, a variable has a specific value or an error would occur during the next program run). You learn how to do this in a program.

Exiting Program Units

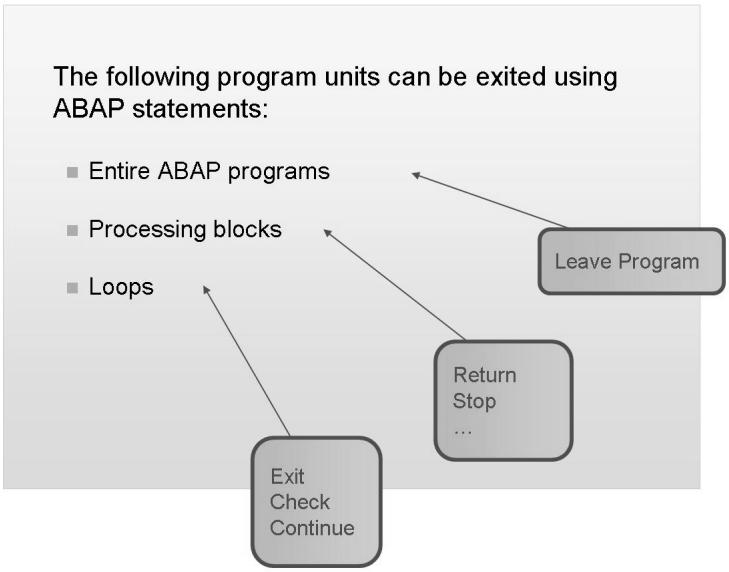


Figure 75: Exiting Program Units

The following program units can be exited using ABAP statements:

- Entire ABAP programs
- Processing blocks
- Loops

Hint: How processing blocks are exited is explained in Unit 5 “Example of Reuse Components in Subroutines”.

Exiting Programs

The LEAVE PROGRAM statement exits the current main program immediately and deletes its internal mode including all loaded programs, instances, and their data. The statement can be inserted anywhere in a PROGRAM in any number of processing blocks. It exits the program irrespective of the program, object, or program group (internal mode) in which it is being executed.

Exiting Processing Blocks

In Unit 5, you will learn how to exit subroutines. Other processing blocks are covered in the course BC400 (ABAP Workbench Foundations).

Exiting Loops

Loops are exited with the following statements:

- Exit
- Check
- Continue

When the **EXIT** statement is executed within a loop, the current loop pass is exited immediately and the program flow continues after the closing statement of the loop.

Hint: Outside of a loop, the **EXIT** statement exits the current processing block. We recommend, however, that you use **EXIT** only within loops.

If the **CHECK** statement is executed within a loop and log_exp is false, the current loop pass is exited immediately and the program flow continues with the next loop pass. For log_exp, any logical expression can be specified.

Hint: Outside of a loop, the **CHECK** statement exits the current processing block.

The **CONTINUE** statement can only be inserted within a loop. It causes the current loop pass to be exited immediately and the program flow to continue with the next loop pass.

Exercise 9: Exiting Program Units (Optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Exit a program loop prematurely

Business Example

You are to implement the "multiplication" function using loop logic and exit the loop as soon as the calculation has been carried out.

Template:

None

Solution:

BC100_REP_LOOP_EXIT

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_LOOP_EXIT**.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one as arithmetic operator of type "C" of one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.
2. Calculate the result dependent on the specified arithmetic operator. Use the "DO" loop for the multiplication and exit it once the calculation has been carried out. Use the **EXIT** command to do this. Use the CASE statement for the case distinction.

Continued on next page

3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

Task 4:

Catch any errors that may occur.

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

Task 5:

Activate and test your program.

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

Solution 9: Exiting Program Units (Optional)

Task 1:

Create a program.

1. Create the executable program **ZBC100_##_LOOP_EXIT**.

   a) Perform this step in the same way as in the previous exercises.

Task 2:

Define four data objects.

1. Define two data objects for integers as operands (suggested names: **gv_int1**, **gv_int2**) and one as arithmetic operator of type "C" of one character in length (suggested name: **gv_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

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

Task 3:

Implement the basic arithmetic operations and output the result in a list.

1. Assign values to the data objects **gv_int1**, **gv_int2**, and **gv_op**.

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

2. Calculate the result dependent on the specified arithmetic operator. Use the "DO" loop for the multiplication and exit it once the calculation has been carried out. Use the **EXIT** command to do this. Use the CASE statement for the case distinction.

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

3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.

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

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

Continued on next page

Task 4:

Catch any errors that may occur.

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

Task 5:

Activate and test your program.

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

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_REP_LOOP_EXIT
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  BC100_REP_LOOP_EXIT.

Data:   gv_int1  TYPE i,
        gv_int2  type i,
        gv_op    TYPE c,
        gv_result type p length 16 decimals 2.

gv_int1 = 3.
gv_int2 = 25.
gv_op   = '*'.
```

Continued on next page

Internal Use SAP Partner Only

```
CASE gv_op.
  WHEN '+'.
    gv_result = gv_int1 + gv_int2.
  WHEN '-'.
    gv_result = gv_int1 - gv_int2.
  WHEN '*'.
    if gv_int1 >= gv_int2.
      do.
        if sy-index > gv_int2.
          exit.
        endif.

        gv_result = gv_result + gv_int1.
      enddo.
    else.
      do.
        if sy-index > gv_int1.
          exit.
        endif.

        gv_result = gv_result + gv_int2.
      enddo.
    endif.
  WHEN '/'.
    if gv_int2 = 0.
      write: / 'Division by zero!'.
    else.
       gv_result = gv_int1 / gv_int2.
    endif.
  When others.
    write: / 'Incorrect Operator!'.
ENDCASE.

write: / 'Result:', gv_result.
```

Lesson Summary

You should now be able to:

- Explain the different ways to exit program units (such as loops) prematurely

Internal Use SAP Partner Only

Unit Summary

You should now be able to:

- Distinguish between the elementary data objects and define and use data objects appropriately
- Describe the logical expressions and relational operators that are needed for branches and case distinction
- Use branches and case distinction correctly
- Explain the basic functions of the Debugger
- Debug simple programs
- Use loops effectively
- Explain the different ways to exit program units (such as loops) prematurely

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 4

Selection Screen, List, and Program Terminations

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Pass data to a program by means of a selection screen and use this data in the program
- Implement an output list with a list header
- Use character string processing effectively
- Recognize program terminations
- Analyze basic errors and resolve them in a program

Unit Contents

Lesson: Selection Screen (Parameters)

Lesson Overview

In this lesson, you learn how to use a selection screen to pass parameters to a program and how to use these parameters within the program.

Lesson Objectives

After completing this lesson, you will be able to:

- Pass data to a program by means of a selection screen and use this data in the program

Business Example

As an employee of "Calculate & Smile", you need to modify your program to add functions that pass data to your program by means of a selection screen. The program is to be able to react dynamically to the data entered and return different results, depending on the input.

Selection screen

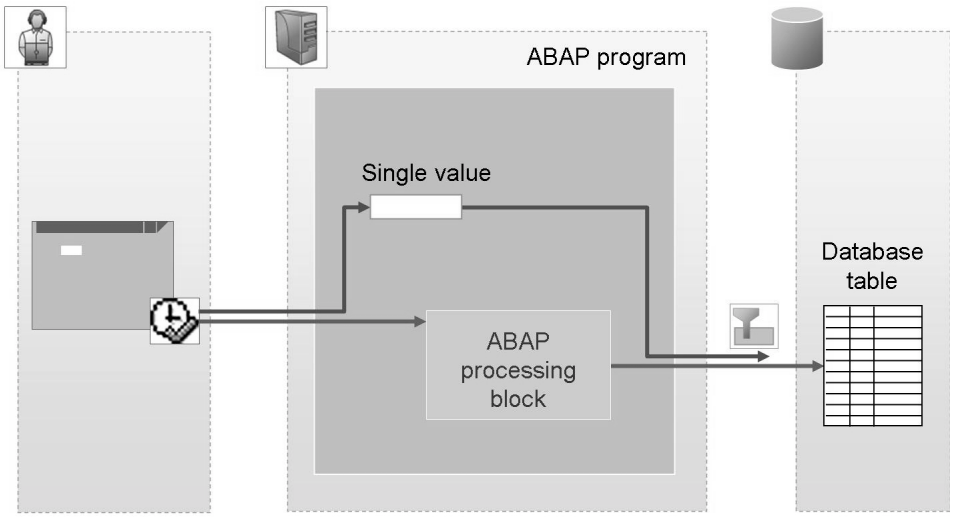


Figure 76: Architecture and Purpose of Selection Screens

In general, selection screens are used for entering selection criteria for data selection. For example, if the program creates a list of data from a very large database table, it often makes sense for users to select the data records they actually require and for the system to read only this data from the database. Apart from reducing the memory requirement, this also reduces the network load.

From a technical perspective, selection screens are ordinary screens. However, they are not designed by the developer directly with the *Screen Painter*, but generated in accordance with declarative statements in the source code.

Advantages of Selection Screens

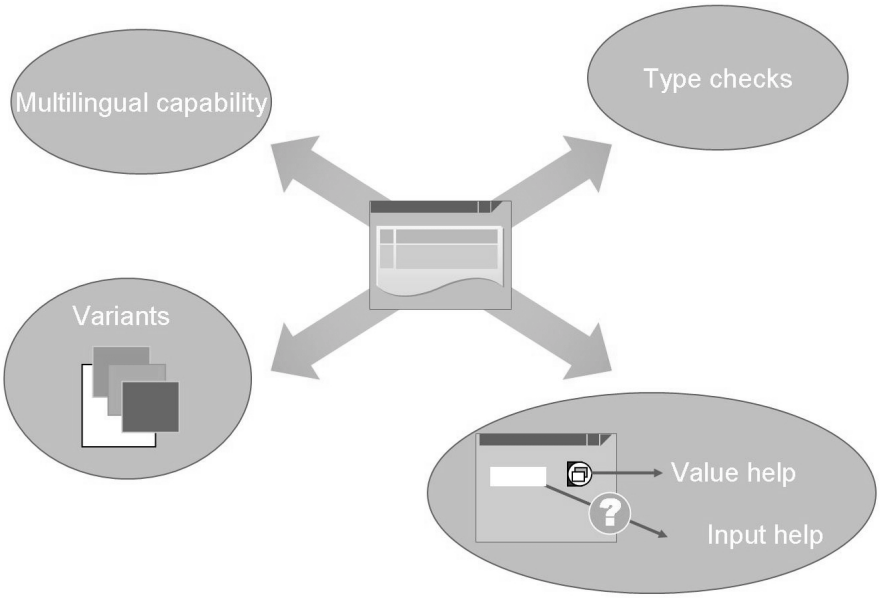


Figure 77: Selection Screen Attributes

The selection screen has the following standard functions:

- Texts on the selection screen (**selection texts**) can be maintained **in several languages**. At runtime, the texts are automatically displayed in the user's logon language.
- The system **checks types** automatically: If the user enters something that does not correspond to the type of the input field, the SAP GUI will ignore it, so that it does not even appear on the selection screen.
- In addition to single value entries (`PARAMETERS`), you can also make **complex selections** (`SELECT-OPTIONS`) on the selection screen. The user can then enter intervals, comparative conditions or even patterns as restrictions. This topic is covered in the BC400 (ABAP Workbench Foundations) course and is not explained at this point.
- If the input field is defined using a Dictionary element (for example, data element), the **field documentation** (documentation of the data element) can be displayed on the input field using the **`F1`** (**input help**) function key.

  The data element search help for **displaying possible inputs** can be called using the **`F4`** (**input help**) function key.
- You can easily save completed selection screens as **variants** for reuse or use in background operation.

Hint: Complex selections (SELECT-OPTIONS) are dealt with in the BC400 (ABAP Workbench Foundations) course.

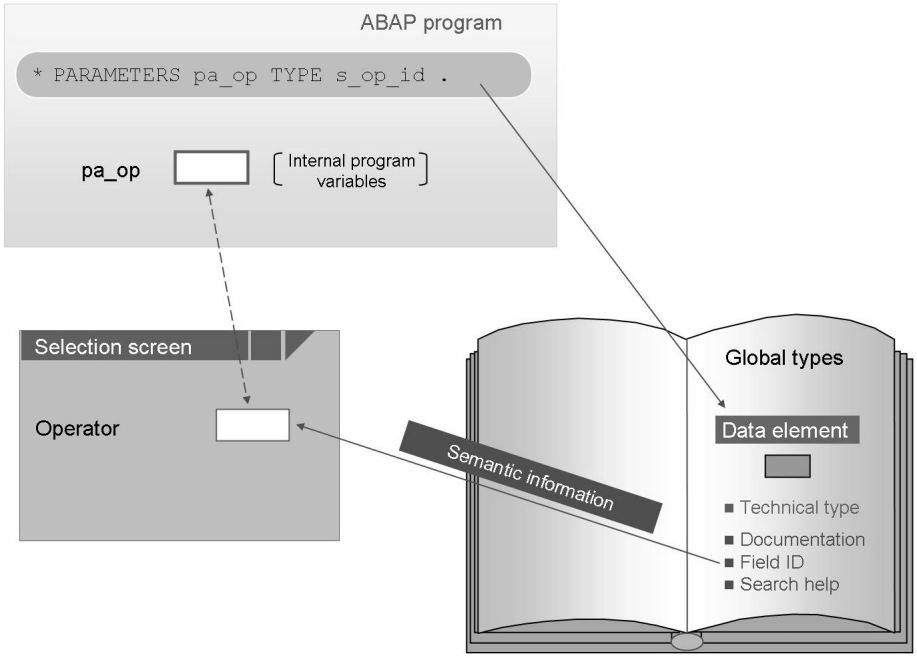


Figure 78: Semantic Information of Global Types on the Selection Screen

If an input field is typed as a **data element**, the following additional semantic information is available on the selection screen:

- The long **field label** of the data element can be adopted to describe the input field on the selection screen (**selection text**) (see next graphic).
- The **documentation** of the data element is automatically available as an **input help** (**`F1`** help).
- If a **search help** is linked to the data element, it is available as **input help** (**`F4`** help).

For more information, refer to the online documentation for the *ABAP Dictionary*.

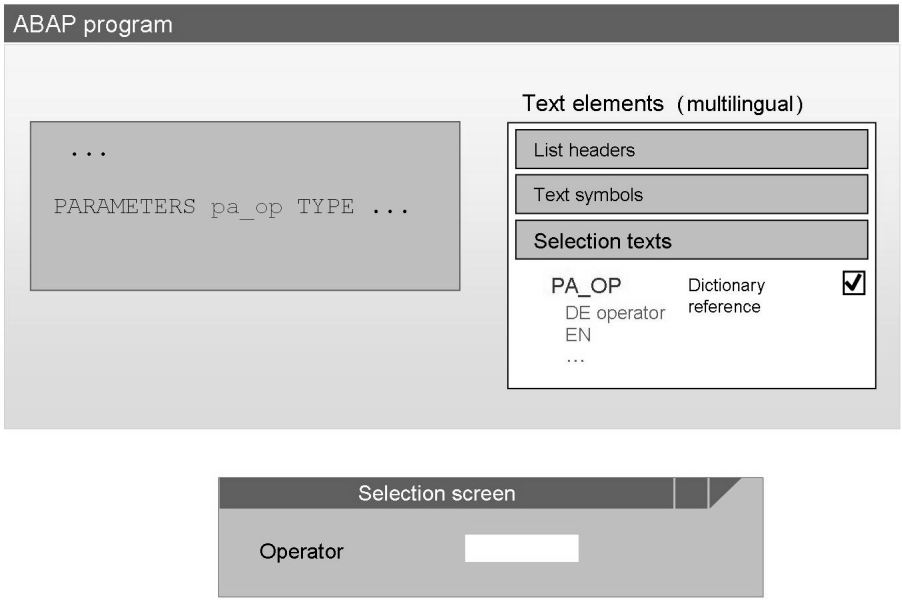


Figure 79: Selection Texts

On the selection screen, the names of the input fields appear as their description by default. However, you can replace these with corresponding **selection texts**, which you can then translate into any further languages you require. At runtime, the selection texts are then displayed in the logon language of the user (automatic language).

Just like the list headers and text symbols, selection texts belong to the text elements of the program. From the *ABAP Editor*, choose *Goto → Text Elements → Selection Texts* to maintain them. You can implement your translation using the menu *Goto → Translation*.

If the input field is typed directly or indirectly as a data element, you can use the **long** field name for the data element ("Dictionary reference") as the selection text. This provides you with an easy option for standardizing texts for all selection screens.

Input Options on Selection Screens

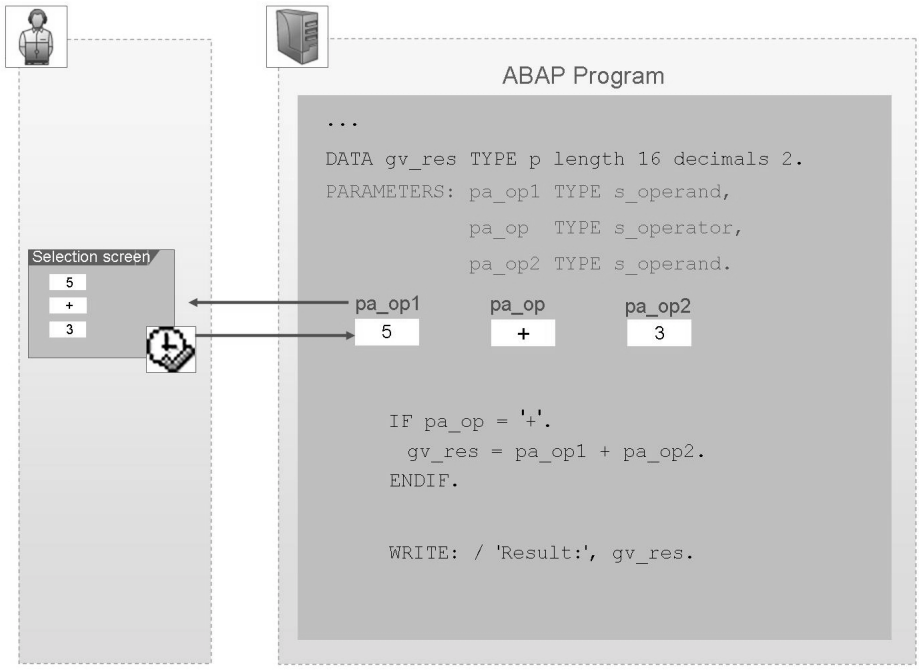


Figure 80: Single-Value Input (PARAMETERS Statement)

The figure above shows how an **input variable** defined using the `PARAMETERS` statement is used and how it is handled at runtime. Once again, the definition of such an input variable creates a variable in the system **and** implicitly generates a selection screen with a corresponding input option.

An input variable is defined in the same way as an ordinary variable. The only difference is that the `PARAMETERS` keyword is used instead of `DATA`.

You have to take three special aspects into consideration:

- The name of the input variable may be up to 8 characters long.
- It may not be typed as the standard types F, STRING, and XSTRING.
- Default values are **not** assigned using the `VALUE` addition, but using the `DEFAULT` addition.

If a default value is assigned by means of the `DEFAULT` addition or the value is assigned **before** the selection screen (INITIALIZATION) is displayed, this value is displayed on the selection screen as a default value that can be overwritten.

If the user enters a value and chooses *Execute*, the input value is transferred to the internal variable and can be used in further calculations or to access database entries, for example.

Exercise 10: Selection Screen

Exercise Objectives

After completing this exercise, you will be able to:

- Use the ABAP statement PARAMETERS to pass data to a program by means of a selection screen

Business Example

You are to enter your operands and operator by means of parameters. You are to use selection texts to label the parameters.

Template:

BC100_UDT_PARAMETER

Solution:

BC100_UDS_PARAMETER

Task 1:

Create a program.

1. Copy the **BC100_UDT_PARAMETER** template and call the new executable program **ZBC100_##_PARAMETER**.

Task 2:

Define three parameters.

1. Define two parameters for integers as operands (suggested names: **pa_int1**, **pa_int2**) and one arithmetic operator of type “C” with one character in length (suggested name: **pa_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

Task 3:

Implement the percentage calculation and output the result to a list.

1. Adjust the CASE statement so that percentage calculation is triggered when “%” is entered as the arithmetic operator.

Task 4:

Catch any errors that may occur.

1. Display an error message in the list if the user tries to divide by zero.

Continued on next page

Task 5:

Activate and test your program.

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

Solution 10: Selection Screen

Task 1:

Create a program.

1. Copy the **BC100_UDT_PARAMETER** template and call the new executable program **ZBC100_##_PARAMETER**.

   a) Perform this step in the same way as in the previous exercises.

Task 2:

Define three parameters.

1. Define two parameters for integers as operands (suggested names: **pa_int1**, **pa_int2**) and one arithmetic operator of type "C" with one character in length (suggested name: **pa_op**). Define a data object for the calculation result (suggested name: **gv_result**). Define it as a packed number that is 16 digits in length and has 2 decimal places. Use the ABAP keyword DATA to do this.

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

Task 3:

Implement the percentage calculation and output the result to a list.

1. Adjust the CASE statement so that percentage calculation is triggered when "%" is entered as the arithmetic operator.

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

Task 4:

Catch any errors that may occur.

1. Display an error message in the list if the user tries to divide by zero.

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

Task 5:

Activate and test your program.

1. Activate your program.

   a) In the *ABAP Editor* toolbar, choose the pushbutton.

2. Test your program.

   a) In the *ABAP Editor* toolbar, choose the pushbutton.

Continued on next page

b) Supply the selection screen parameters with values and choose .

c) Check the result. Test the message output in the event of an error too.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_UDS_PARAMETER
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_uds_parameter.

PARAMETER: pa_int1  TYPE i DEFAULT 13,
           pa_op    TYPE c DEFAULT '%',
           pa_int2  TYPE i DEFAULT 130.

DATA:      gv_result TYPE p LENGTH 16 DECIMALS 2.

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    IF pa_int2 = 0.
      WRITE: / 'Division durch Null!'.
    ELSE.
      gv_result = pa_int1 / pa_int2.
    ENDIF.
  WHEN '%'.
    IF pa_int2 = 0.
      WRITE: / 'Division by zero!'.
    ELSE.
      gv_result = pa_int1 * 100 / pa_int2.
    ENDIF.
  WHEN OTHERS.
    WRITE: / 'Incorrect Operator!'.
ENDCASE.

WRITE: / 'Result:', gv_result.
```

Lesson Summary

You should now be able to:

- Pass data to a program by means of a selection screen and use this data in the program

Lesson: Lists

Lesson Overview

This lesson focuses on the classic ABAP list.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement an output list with a list header

Business Example

As an employee of "Calculate & Smile", you want to use list headers and text literals to output a result. The layout of the list is to be adjusted as required for the output.

ABAP List

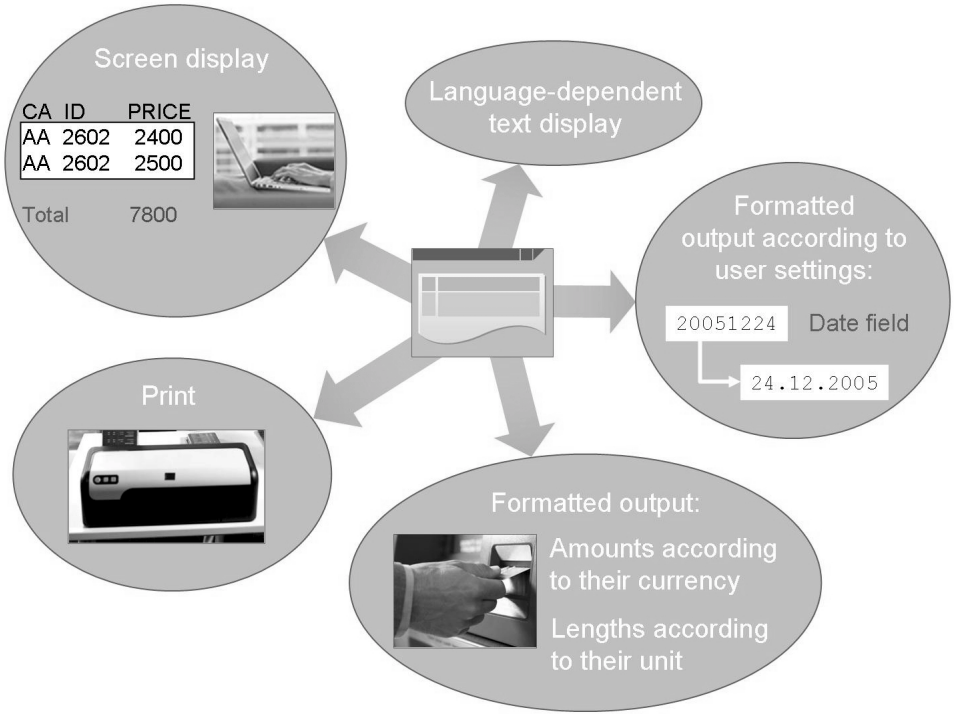


Figure 81: Features of Lists

The purpose of a list is to output data with the least possible programming effort. Lists also take the special requirements of business data into account:

- Lists can be designed for a number of languages: Texts and headers appear in the logon language whenever a corresponding translation is available.
- Lists can display monetary values in the appropriate currency.

The following options are available when outputting a list:

- Screen: You can add colors and icons here.
- Printer
- Internet or intranet: The system automatically converts the list to HTML format for this.
- Save: You can save lists within the SAP system as well as outside (for further processing, for example, using spreadsheet programs).

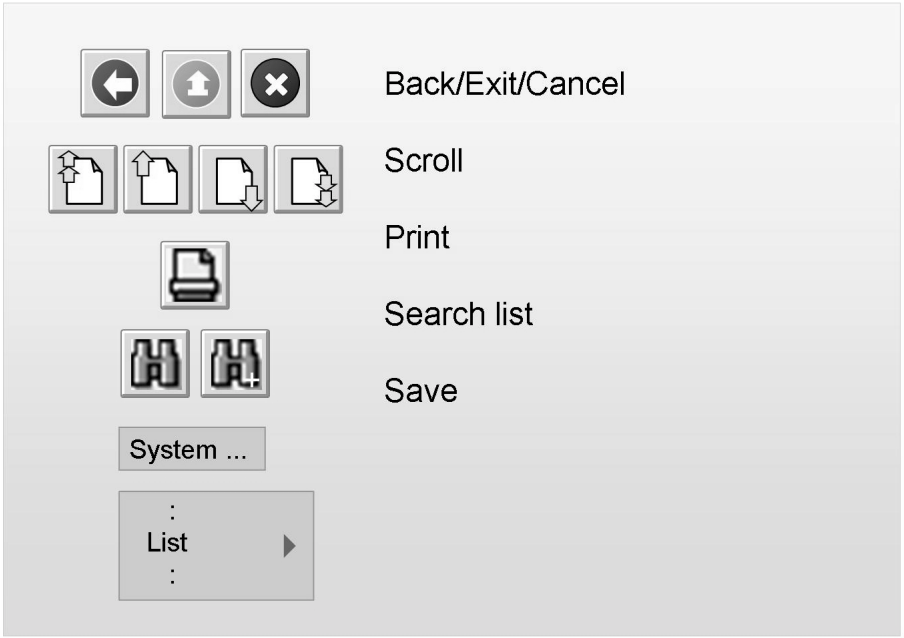


Figure 82: Standard List Functions

The standard user interface for a list offers a range of functions.

You can use the *Menu Painter* to adapt the default list interface to your own needs. More information is available in the documentation for this tool or in the corresponding training course.

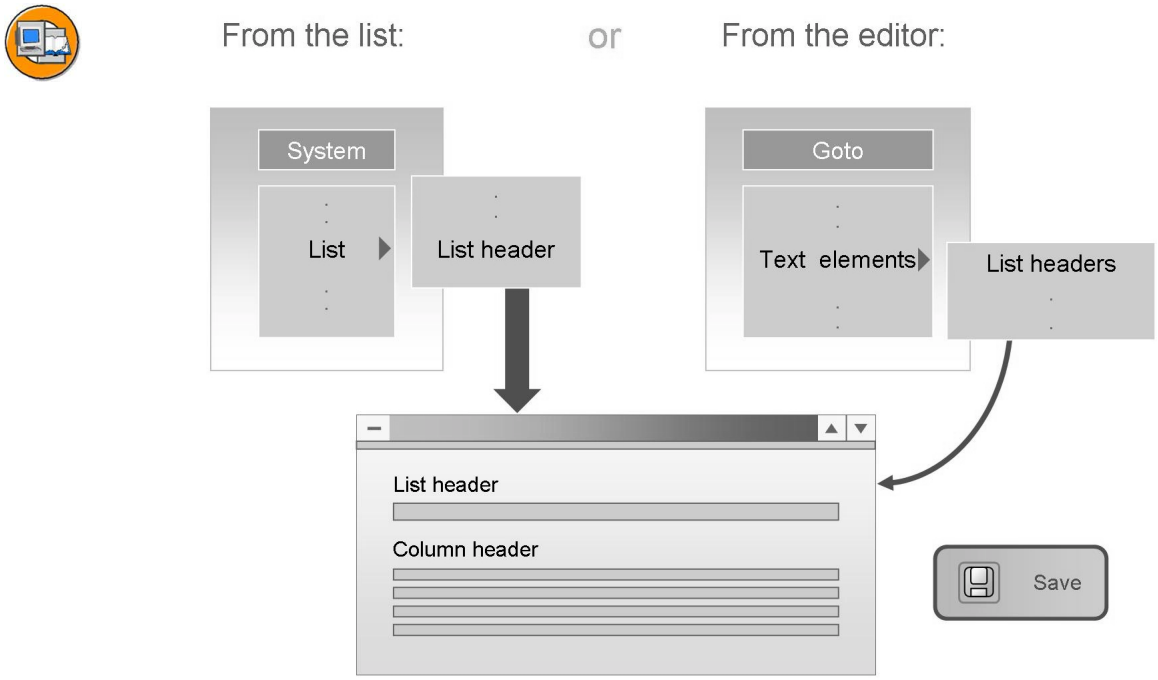


Figure 83: List and Column Headers

In an earlier lesson, you learned how to implement translatable **text symbols** so that the interface of your program is displayed in the logon language of the user. Text symbols are available in all types of programs.

For executable programs (reports) in particular, you have the option of maintaining a single line **list header** and up to four rows of **column headers** in addition to the text symbols for the classic ABAP list.

Together with text symbols and selection texts, list and column headers belong to the **text elements** of a program.

To **maintain** the headers, you first have to activate your program. You then create the list by executing the program. You can then maintain the headers directly above your list by choosing *System* → *List* → *List Header* in the menu. The next time you start the program, they will appear in the list automatically.

If you want to change the headers you maintained, there are two approaches you can take. You can start the program, generate the list, and maintain the headers as previously outlined. Alternatively, you can access the maintenance screen for changing headers from the editor in which the program was loaded by choosing *Goto* → *Text Elements* → *Selection Texts*.

To translate list and column headers from within the *ABAP Editor*, choose *Goto* → *Translation*.

Exercise 11: List and List Headers

Exercise Objectives

After completing this exercise, you will be able to:

- Use translatable text elements on the selection screen and in the ABAP list
- Use icons and colors in a classic list

Business Example

You want to make the design of the selection screen and the ABAP list more appealing by using translatable texts, colors, and icons.

Template:

BC100_RPT_REP

Solution:

BC100_RPS_REP

Task 1:

Create a program.

1. Copy the **BC100_RPT_REP** template and call the new executable program **ZBC100_##_REP**.

Task 2:

Declare the type group “ICON”.

1. Declare the type group **ICON**. Use the ABAP keyword TYPE-POOLS to do this.

Task 3:

Display an icon (calculator), the input parameters, the operator, and the result in an ABAP list.

1. Display the “ICON_CALCULATION” icon in the output. Use the pattern function to do this.
2. Include the input parameters, the operator, and the result in the output. Highlight the result in color. Use **COL_TOTAL** as the color. Use the ABAP keyword **COLORS** and the **WRITE** statement to do this.
3. Create a list header. To do this, follow the description in the training material. Improve the appearance of the output list by displaying the output elements at a suitable point.

Continued on next page

Task 4:

Activate and test your program.

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

Solution 11: List and List Headers

Task 1:

Create a program.

1. Copy the **BC100_RPT_REP** template and call the new executable program **ZBC100_##_REP**.

   a) Perform this step in the same way as in the previous exercises.

Task 2:

Declare the type group “ICON”.

1. Declare the type group **ICON**. Use the ABAP keyword TYPE-POOLS to do this.

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

Task 3:

Display an icon (calculator), the input parameters, the operator, and the result in an ABAP list.

1. Display the “ICON_CALCULATION” icon in the output. Use the pattern function to do this.

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

2. Include the input parameters, the operator, and the result in the output. Highlight the result in color. Use **COL_TOTAL** as the color. Use the ABAP keyword **COLORS** and the **WRITE** statement to do this.

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

3. Create a list header. To do this, follow the description in the training material. Improve the appearance of the output list by displaying the output elements at a suitable point.

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

Task 4:

Activate and test your program.

1. Activate your program.

   a) In the *ABAP Editor* toolbar, choose the pushbutton.

2. Test your program.

   a) In the *ABAP Editor* toolbar, choose the pushbutton.

Continued on next page

b) Supply the selection screen parameters with values and choose .

c) Check the result. Test the message output in the event of an error too.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_REP
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  BC100_RPS_REP.

Type-pools: icon.

Parameter: pa_int1  TYPE i default 13,
           pa_op    TYPE c default '%',
           pa_int2  type i default 130.

Data:      gv_result type p length 16 decimals 2.

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    if pa_int2 = 0.
      WRITE: / 'Division by zero!'.
    else.
      gv_result = pa_int1 / pa_int2.
    endif.
  when '%'.
    if pa_int2 = 0.
    WRITE: / 'Division by zero!'.
    else.
      gv_result = pa_int1 * 100 / pa_int2.
    endif.
  When others.
    WRITE:  / 'Incorrect operator!'.
ENDCASE.
```

Continued on next page


```
write: / ICON_CALCULATION AS ICON,
         pa_int1,
         22 pa_op,
         pa_int2,
         '=',
         gv_result color col_total.
```


Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Implement an output list with a list header

Lesson: Processing Character Strings

Lesson Overview

This lesson describes the statements available to process character strings in character-like data objects.

Lesson Objectives

After completing this lesson, you will be able to:

- Use character string processing effectively

Business Example

As an employee of "Calculate & Smile", you need to learn about character string processing and how you can use it to read and change data from data objects at runtime. By using the functions available, you can search for and modify variables more easily.

Processing Character Strings

CONCATENATE	To append fields
FIND	To search for contents
REPLACE	To replace contents
SPLIT	To divide up fields
…	

Figure 84: Processing Character Strings

Character-like data objects contain strings (or character strings). A character-like data object can contain a character-like data type (c, d, n, t, string).

Concatenating Character Strings (CONCATENATE)

```
REPORT  BC100_RPS_CONCATENATE.

PARAMETERS: pa_fname(30)   type c,
            pa_lname(30)   type c.

DATA: gv_name              type string,
      gv_sep               type c          value ' '.

CONCATENATE pa_fname pa_lname INTO gv_name SEPARATED BY gv_sep.

WRITE: / 'Name:', gv_name.

WRITE: / 'Name:', pa_fname, pa_lname.
```

Name: Hugo Müller

Name: Hugo Müller

Figure 85: Concatenating Character Strings (CONCATENATE)

If you want to create one character string from two separate character strings, you can use the **CONCATENATE** statement. You can insert a separator between the two data objects. You can do this using the **SEPARATED BY** addition. The content of one data object **GV_SEP** is inserted between the other data objects. If the target field is shorter than the combined length of the source fields, the length of the new character string is restricted to the length of the target field. It is therefore recommended to use the “string” data type for this type of operation. The “string” data type allows you to assign the required length to the target string.

Hint: For more information, see the system documentation. Search for the term “Byte and Character String Processing”.

Searching (FIND) and Replacing (REPLACE) Within Character Strings

```
REPORT  bc100_rps_find_replace.

DATA: gv_text TYPE string VALUE 'Introduction to Programming (with JAVA)'.

FIND FIRST OCCURRENCE
       OF SUBSTRING 'JAVA' IN gv_text
       IGNORING CASE.

IF sy-subrc = 0.
  REPLACE ALL OCCURRENCES
             OF SUBSTRING 'JAVA' IN gv_text
             WITH 'ABAP'
             RESPECTING CASE.
ENDIF.

MESSAGE gv_text TYPE 'I'.
```

Introduction to Programming (with ABAP)

Figure 86: Searching (FIND) and Replacing (REPLACE) Within Character Strings

This section deals with two important statements for processing character strings: **FIND** and **REPLACE**. To search for a value in a character string, you use **FIND**, and to replace a value, you use **REPLACE**. The **FIND** and **REPLACE** statements outlined here are based on similar principles. Whereas **FIND** simply searches for and reports on one or all occurrences of a subsequence or pattern in a data object, **REPLACE** also replaces any occurrences found with the content specified.

Hint: For more information, see the system documentation. Search for the term “Byte and Character String Processing”.

Splitting Character Strings (SPLIT)

```
REPORT  bc100_rps_split.

DATA: gv_text     TYPE string VALUE 'Hugo Müller',
      gv_fname TYPE string,
      gv_lname TYPE string.

SPLIT gv_text AT ' ' INTO gv_fname gv_lname.


WRITE: / 'Firstname:', gv_fname.
WRITE: / 'Lastname:',  gv_lname.
```

Firstname: Hugo
Lastname: Müller

Figure 87: Splitting Character Strings (SPLIT)

Using the **SPLIT** statement, you can divide character strings into substrings. The character string is split before and after the separator and is then divided up into individual target fields. If the number of target fields available is not sufficient, the remaining part of the source character string, including the separator, is entered into the last target field.

Hint: For more information, see the system documentation. Search for the term “Byte and Character String Processing”.

Exercise 12: Processing Character Strings (optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Read and modify the contents of data objects at runtime by using character string functions

Business Example

The “CONCATENATE”, “FIND”, “REPLACE”, and “SPLIT” character string functions are to be used to read and modify the contents of data objects at runtime. By using the character string functions available, you can search within and modify data objects more easily.

Template:

None

Solution:

BC100_RPS_CONCATENATE

BC100_RPS_FIND_REPLACE

BC100_RPS_SPLIT

Task 1:

Create a program.

1. Create a new program and call the new executable program **ZBC100_##_CONCATENATE**.

Task 2:

Define two parameters.

1. Define two parameters for the first and last names as type “Char 30” (suggested names: **pa_fnam** and **pa_lnam**). Define a data object for the result (suggested name: **gv_name**). Define it as a “STRING”. Use the ABAP keyword DATA to do this.

Continued on next page

Task 3:

Assign fields to a target field by using the character string function **CONCATENATE**.

1. Use the **CONCATENATE** character string function to assign the parameters “pa_fnam” and “pa_lnam” to the target field “gv_name”. Separate the two parameters with a separator.

Task 4:

Activate and test your program.

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

Task 5:

If you have time: Create another program.

1. Create a new program and call the new executable program **ZBC100_##_FIND_REPLACE**.

Task 6:

Define a data object.

1. Define a data object **gv_text** of type “STRING”. Use the ABAP keyword DATA to do this. Assign the text “Introduction to Programming (with JAVA)” to the data object. Use the “VALUE” keyword to do this.

Task 7:

Use the character string function **FIND** to search for occurrences of the term “JAVA”.

1. Use the character string function **FIND** to search for the first occurrence of the term “JAVA”.
2. If the term is found, replace it with the term “ABAP”. Respect the capitalization.
3. Output the result as a message statement of type “I”.

Task 8:

Activate and test your program.

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

Continued on next page

Task 9:

If you have time: Create another program.

1. Create a new program and call the new executable program **ZBC100_##_SPLIT**.

Task 10:

Define three data objects.

1. Define the three data objects **gv_text**, **gv_fname**, and **gv_lnam** as type "STRING". Use the ABAP keyword DATA to do this. Assign the text "Hugo Müller" to the data object **gv_text**. Use the "VALUE" keyword to do this.

Task 11:

Split up the data object "gv_text" by using the character string function **SPLIT**.

1. Split up the data object "gv_text" by using the character string function **SPLIT**, and assign the contents to the data objects "gv_fname" and "gv_lname". Separate the character string where the space occurs.
2. Output the data objects "gv_fname" and "gv_lname" as an ABAP list. Use the **WRITE** statement to do this.

Task 12:

Activate and test your program.

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

Solution 12: Processing Character Strings (optional)

Task 1:

Create a program.

1. Create a new program and call the new executable program **ZBC100_##_CONCATENATE**.
   a) Perform this step in the same way as in the previous exercises.

Task 2:

Define two parameters.

1. Define two parameters for the first and last names as type “Char 30” (suggested names: **pa_fnam** and **pa_lnam**). Define a data object for the result (suggested name: **gv_name**). Define it as a “STRING”. Use the ABAP keyword DATA to do this.
   a) See source code excerpt 1 from the model solution.

Task 3:

Assign fields to a target field by using the character string function **CONCATENATE**.

1. Use the **CONCATENATE** character string function to assign the parameters “pa_fnam” and “pa_lnam” to the target field “gv_name”. Separate the two parameters with a separator.
   a) See the source code excerpt from the model solution.

Task 4:

Activate and test your program.

1. Activate your program.
   a) Carry out this step in the usual manner.
2. Test your program.
   a) Carry out this step in the usual manner.
   b) Check the result. Test the message output in the event of an error too.

Continued on next page

Task 5:

If you have time: Create another program.

1. Create a new program and call the new executable program **ZBC100_##_FIND_REPLACE**.

   a) Perform this step in the same way as in the previous exercises.

Task 6:

Define a data object.

1. Define a data object **gv_text** of type “STRING”. Use the ABAP keyword DATA to do this. Assign the text “Introduction to Programming (with JAVA)” to the data object. Use the “VALUE” keyword to do this.

   a) See source code excerpt 2 from the model solution.

Task 7:

Use the character string function **FIND** to search for occurrences of the term “JAVA”.

1. Use the character string function **FIND** to search for the first occurrence of the term “JAVA”.

   a) See the source code excerpt from model solution 2.

2. If the term is found, replace it with the term “ABAP”. Respect the capitalization.

   a) See the source code excerpt from model solution 2.

3. Output the result as a message statement of type “I”.

   a) See the source code excerpt from model solution 2.

Task 8:

Activate and test your program.

1. Activate your program.

   a) Carry out this step in the usual manner.

2. Test your program.

   a) Carry out this step in the usual manner.

   b) Check the result. Test the message output in the event of an error too.

Continued on next page

Task 9:

If you have time: Create another program.

1. Create a new program and call the new executable program **ZBC100_##_SPLIT**.

   a) Perform this step in the same way as in the previous exercises.

Task 10:

Define three data objects.

1. Define the three data objects **gv_text**, **gv_fname**, and **gv_lnam** as type "STRING". Use the ABAP keyword DATA to do this. Assign the text "Hugo Müller" to the data object **gv_text**. Use the "VALUE" keyword to do this.

   a) See source code excerpt 3 from the model solution.

Task 11:

Split up the data object "gv_text" by using the character string function **SPLIT**.

1. Split up the data object "gv_text" by using the character string function **SPLIT**, and assign the contents to the data objects "gv_fname" and "gv_lname". Separate the character string where the space occurs.

   a) See the source code excerpt from model solution 3.

2. Output the data objects "gv_fname" and "gv_lname" as an ABAP list. Use the **WRITE** statement to do this.

   a) See the source code excerpt from model solution 3.

Task 12:

Activate and test your program.

1. Activate your program.

   a) Carry out this step in the usual manner.

2. Test your program.

   a) Carry out this step in the usual manner.

   b) Check the result. Test the message output in the event of an error too.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_CONCATENATE
*&
*&---------------------------------------------------------------------*
```

Continued on next page


```
*&
*&
*&---------------------------------------------------------------------*

REPORT  BC100_RPS_CONCATENATE.

parameters: pa_fnam(30) type c default 'Hugo',
            pa_lnam(30) type c default 'Müller'.

data: gv_name type string,
      gv_sep  type c value ' '.

concatenate pa_fnam pa_lnam into gv_name separated by gv_sep.

write: / 'Name:', gv_name.

write: / 'Name:', pa_fnam, pa_lnam.
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_FIND_REPLACE
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_rps_find_replace.

DATA: gv_text TYPE string VALUE 'Introduction to Programming (with JAVA)'.

FIND FIRST OCCURRENCE OF SUBSTRING 'JAVA' IN gv_text IGNORING CASE.

IF sy-subrc = 0.
  REPLACE ALL OCCURRENCES OF SUBSTRING 'JAVA' IN gv_text WITH 'ABAP' RESPECTING CASE.
ENDIF.

MESSAGE gv_text TYPE 'I'.
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_CONCATENATE
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_rps_find_replace.
```

Continued on next page


```
DATA: gv_text  TYPE string VALUE 'Hugo Müller',
      gv_fname TYPE string,
      gv_lname TYPE string.

SPLIT gv_text AT ' ' INTO gv_fname gv_lname.


WRITE: / 'Firstname:', gv_fname.
WRITE: / 'Lastname:',  gv_lname.
```

Lesson Summary

You should now be able to:

- Use character string processing effectively

Internal Use SAP Partner Only

Lesson: Program Terminations

Lesson Overview

In this lesson, you learn how to recognize runtime errors that are not caught. You use transaction ST22 (ABAP Dump Analysis) to analyze errors.

Lesson Objectives

After completing this lesson, you will be able to:

- Recognize program terminations
- Analyze basic errors and resolve them in a program

Business Example

As an employee of "Calculate & Smile", you need to react to program terminations. You familiarize yourself with the tool for displaying runtime errors (program terminations), identify the location of the error in the program terminated, analyze the problem, and finally resolve the problem in your program to ensure that it then runs without errors.

Runtime Errors

Static checks cannot fully eliminate runtime errors. For example, program functions in which the arguments of statements are specified dynamically as field content cannot be checked statically. If the runtime environment encounters an error when the program is executed, it triggers a runtime error.

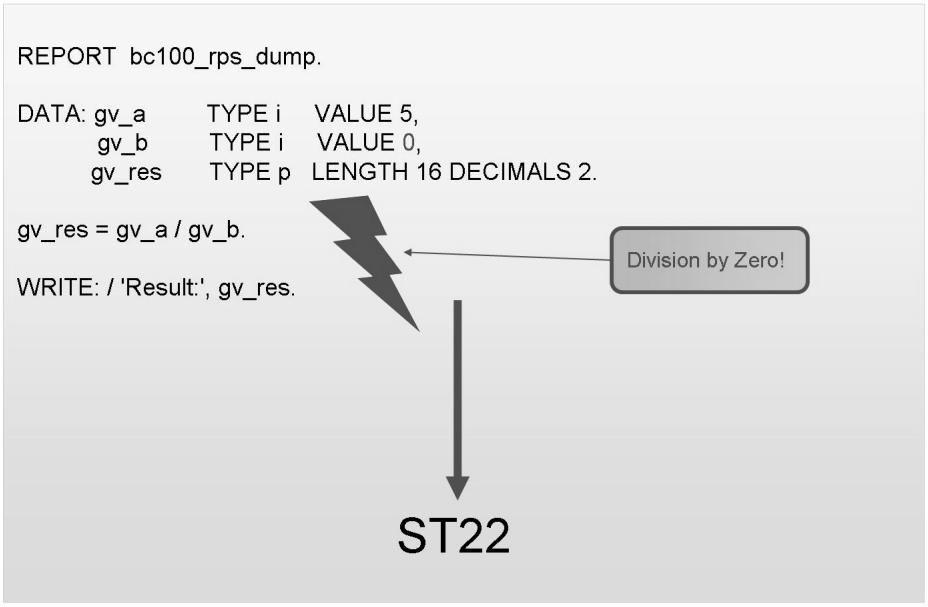


Figure 88: Triggering a Runtime Error

If a runtime error is not caught, the runtime environment terminates the program immediately, and generates and displays a short dump (in transaction ST22).

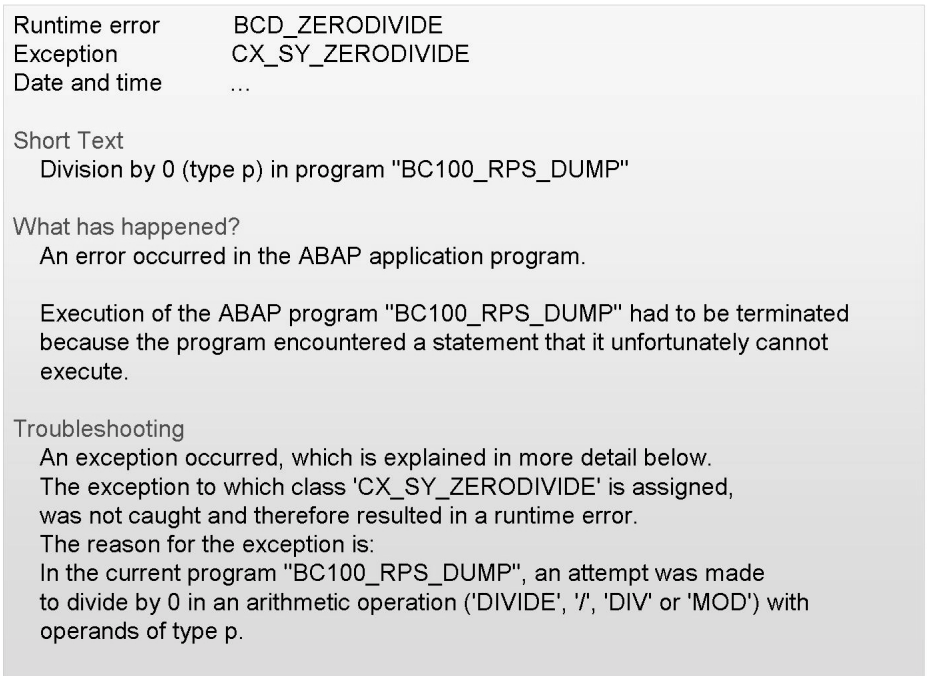


Figure 89: Short Dumps Triggered by Runtime Errors

A short dump is divided into different sections that document the error. The short dump overview indicates which information is displayed, for example, the contents of data objects, active calls, control structures, and so on.

```
Missing Handling of System Exception
   Program                  BC100_RPS_DUMP
Trigger Location of Exception
   Program                  BC100_RPS_DUMP
   Include                  BC100_RPS_DUMP
   Line                     15
   Module name              START-OF-SELECTION

  1 *&---------------------------------------------------------------------*
  2 *& Report  BC100_RPS_DUMP
  3 *&
  4 *&---------------------------------------------------------------------*
  5 *&
  6 *&
  7 *&---------------------------------------------------------------------*
  8
  9 REPORT  bc100_rps_dump.
 10
 11 DATA: gv_a TYPE i VALUE 5,
 12       gv_b TYPE i VALUE 0,
 13       gv_res TYPE p LENGTH 16 DECIMALS 2.
 14
>>>>> gv_res = gv_a / gv_b.
 16
 17 WRITE: / 'Result:', gv_res.
```

Figure 90: Short Dumps Triggered by Runtime Errors 2

From the short dump display, you can branch to the ABAP Debugger, which places the cursor at the point in the code where the program was terminated. To do this, choose the “Debugger” pushbutton in transaction ST22.

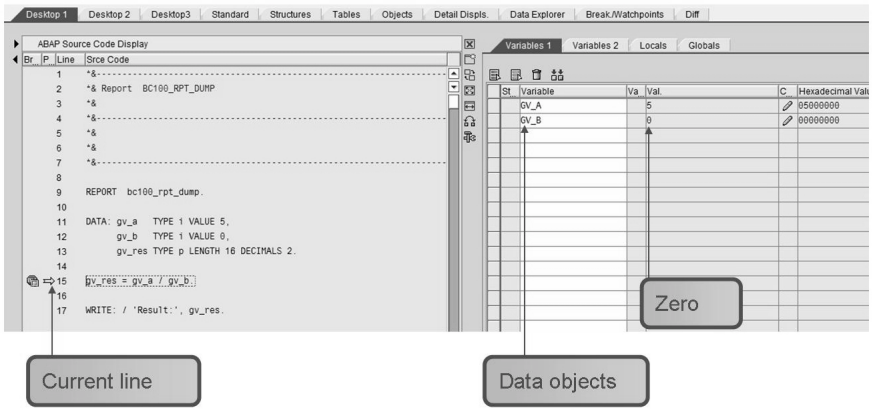


Figure 91: Calling the Debugger from Transaction ST22

By default, short dumps are stored in the system for 14 days. The transaction for managing short dumps is ST22. Using the “Keep” function, short dumps can be retained indefinitely.

Exercise 13: Analyzing and Resolving Program Errors

Exercise Objectives

After completing this exercise, you will be able to:

- Analyze a program termination by using transaction ST22 and then resolve the error

Business Example

When a program terminates, you use transaction ST22 to analyze the termination. You identify the location of the error and determine the cause. You then resolve the error in the program.

Template:

BC100_RPT_DUMP

BC100_RPT_DUMP2

BC100_RPT_DUMP3

Solution:

BC100_RPS_DUMP

BC100_RPS_DUMP2

BC100_RPS_DUMP3

Task 1:

Copy a program and then execute it.

1. Copy the program **BC100_RPT_DUMP** and call the new executable program **ZBC100_##_DUMP**.

Task 2:

Activate and test your program.

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

Task 3:

Locate and resolve the error.

1. Analyze the error message and the point in the code where the program was terminated.

Continued on next page

2. Resolve the error in your program. Then activate the program and test it again.

Result

Refer to the slides from this lesson and the sample solution to help you troubleshoot and resolve the error.

Task 4:

If you have time: Copy another program and then execute it.

1. Copy the program **BC100_RPT_DUMP2** and call the new executable program **ZBC100_##_DUMP2**.

Task 5:

Activate and test your program.

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

Task 6:

If you have time: Locate and resolve the error.

1. Analyze the error message and the point in the code where the program was terminated.
2. Resolve the error in your program. Then activate the program and test it again.

Result

Refer to the slides from this lesson and sample solution no. 2 to help you troubleshoot and resolve the error.

Solution 13: Analyzing and Resolving Program Errors

Task 1:

Copy a program and then execute it.

1. Copy the program **BC100_RPT_DUMP** and call the new executable program **ZBC100_##_DUMP**.
   a) Perform this step in the same way as in the previous exercises.

Task 2:

Activate and test your program.

1. Activate your program.
   a) Carry out this step in the usual manner.
2. Test your program.
   a) Carry out this step in the usual manner.
   b) The program terminates before it is fully executed. Use transaction ST22 to view the error message.

Task 3:

Locate and resolve the error.

1. Analyze the error message and the point in the code where the program was terminated.
   a) View the error message in detail.
   b) Execute your program in debugging mode and display the contents of the data objects. If required, change the contents if they are causing the program to terminate.
2. Resolve the error in your program. Then activate the program and test it again.
   a) Refer to the sample solution for details.

Result

Refer to the slides from this lesson and the sample solution to help you troubleshoot and resolve the error.

Continued on next page

Task 4:

If you have time: Copy another program and then execute it.

1. Copy the program **BC100_RPT_DUMP2** and call the new executable program **ZBC100_##_DUMP2**.
   a) Perform this step in the same way as in the previous exercises.

Task 5:

Activate and test your program.

1. Activate your program.
   a) Carry out this step in the usual manner.
2. Test your program.
   a) Carry out this step in the usual manner.
   b) The program terminates before it is fully executed. Use transaction ST22 to view the error message.

Task 6:

If you have time: Locate and resolve the error.

1. Analyze the error message and the point in the code where the program was terminated.
   a) View the error message in detail.
   b) Execute your program in debugging mode and display the contents of the data objects. If required, change the contents if they are causing the program to terminate.
2. Resolve the error in your program. Then activate the program and test it again.
   a) Refer to the sample solution for details.

Result

Refer to the slides from this lesson and sample solution no. 2 to help you troubleshoot and resolve the error.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_DUMP
*&
*&---------------------------------------------------------------------*
```

Continued on next page

Internal Use SAP Partner Only

```
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_rps_dump.

DATA: gv_a   TYPE i VALUE 5,
      gv_b   TYPE i VALUE 0,
      gv_res TYPE p LENGTH 16 DECIMALS 2.

IF gv_b > 0.
  gv_res = gv_a / gv_b.
ELSE.
  WRITE: / 'Division by zero!'.
ENDIF.

WRITE: / 'Result:', gv_res.
*&---------------------------------------------------------------------*
*& Report  BC100_RPS_DUMP2
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_rps_dump2.

DATA: gv_a   TYPE i VALUE 300,
      gv_b   TYPE i VALUE 400,
      gv_res TYPE p LENGTH 16.

gv_res = gv_a * gv_b.

WRITE: / 'Result:', gv_res.
```


Internal Use SAP Partner Only

Lesson Summary

You should now be able to:

- Recognize program terminations
- Analyze basic errors and resolve them in a program

Unit Summary

You should now be able to:

- Pass data to a program by means of a selection screen and use this data in the program
- Implement an output list with a list header
- Use character string processing effectively
- Recognize program terminations
- Analyze basic errors and resolve them in a program

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 5

Example of Reuse Components in Subroutines

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Describe the purpose of using reuse components

Unit Contents

Lesson: Purpose of Reuse Components

Lesson Overview

In this lesson, you will learn why it makes sense to place parts of programs in subroutines.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of using reuse components

Business Example

As an employee of "Calculate & Smile", you need to learn about the purpose of reuse components since moving identical source code blocks to a subroutine minimizes maintenance effort and makes it easier to read a program.

Modularization Technique

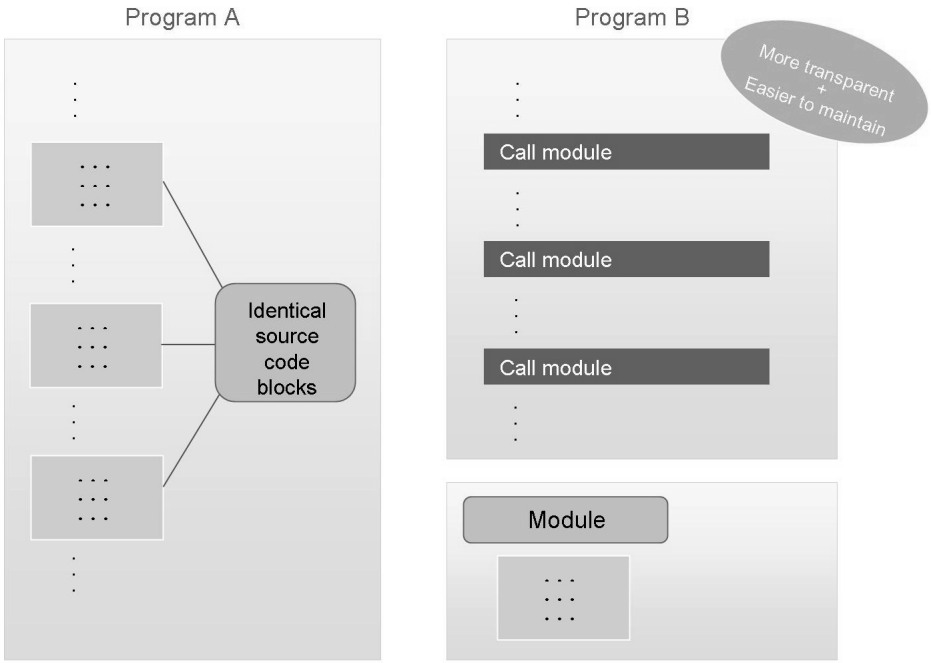


Figure 92: Options for Using Modularization Units

Options for Using Modularization Units

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

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

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

Local Program Modularization

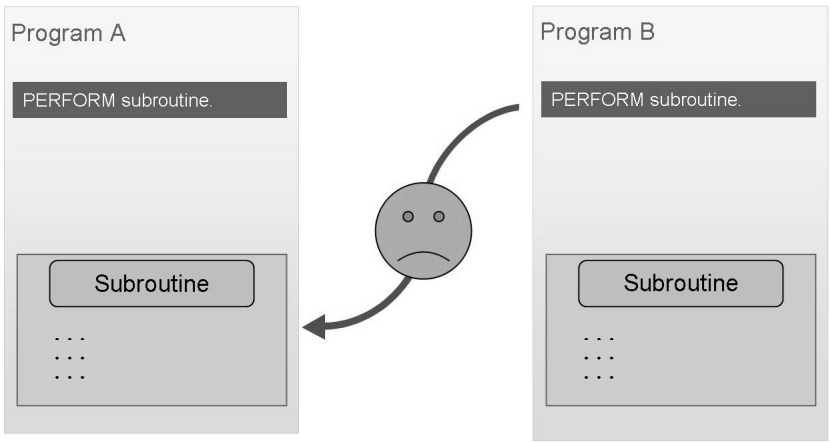


Figure 93: Local Program Modularization

For local program modularization, **subroutines** (also called **form routines**) are available in the ABAP programming language.

With this local modularization technique, modularization units are only to be used in the program in which they were implemented. To call the local module, no other program must be loaded to the user context at runtime in this case. Subroutines

can have the same name in different programs without this resulting in conflicts. This is because the source code for the programs is handled separately in the main memory of the application server.

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

Encapsulating Data

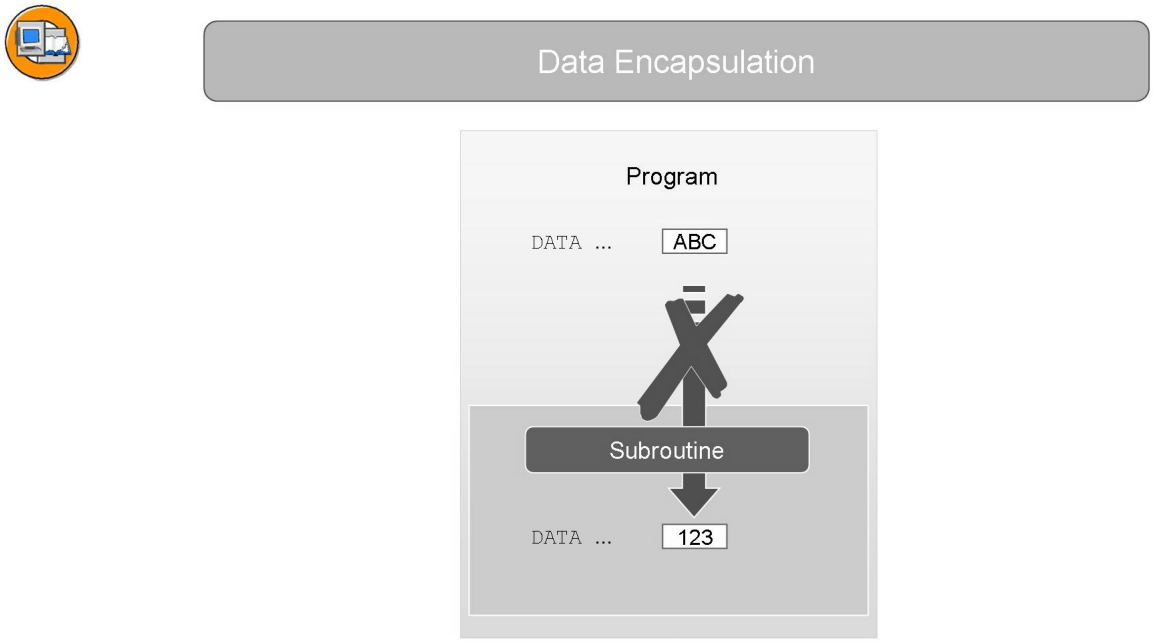


Figure 94: Separating Data

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

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

Data Transports, Parameters, and Interface

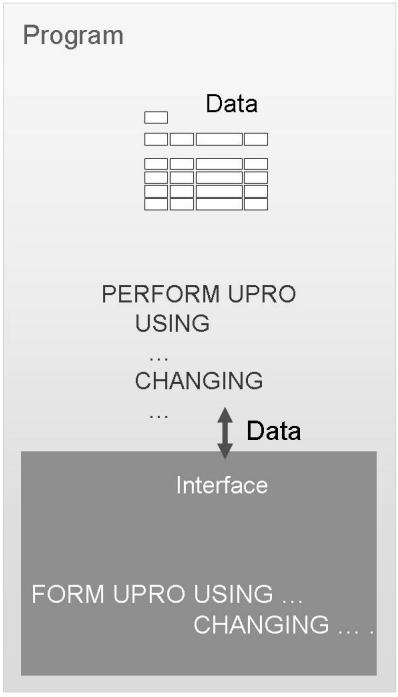


Figure 95: Data Transports Between the Program and the Modularization Unit

Parameters are **used** to **exchange data** between the program and the subroutine. All the parameters of a subroutine are called its **parameter interface (or interface)**. The parameters available are specified when you define the subroutine. You distinguish between parameters based on whether they are used for passing data to the subroutine (**using parameter**) or for passing data to the modularization unit, where it is then changed and returned (**changing parameter)**.

Lesson Summary

You should now be able to:

- Describe the purpose of using reuse components

Unit Summary

You should now be able to:

- Describe the purpose of using reuse components

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 6

Appendix

Internal Use SAP Partner Only

Unit Overview

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

Unit Objectives

After completing this unit, you will be able to:

- Develop simple subroutines
- Pass parameters to these subroutines
- Exit subroutines prematurely
- Debug subroutines

Unit Contents

Lesson: Calling Subroutines and Passing Parameters

Lesson Overview

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

Lesson Objectives

After completing this lesson, you will be able to:

- Develop simple subroutines
- Pass parameters to these subroutines

Business Example

As an employee of "Calculate & Smile", you want to place functions in subroutines so that you can also use these from other parts of the program. Parameters are to be passed to the subroutine and the result is to be returned to the main program. This subroutine technique makes the program more transparent and easier to maintain.

Internal Program Modularization with Subroutines

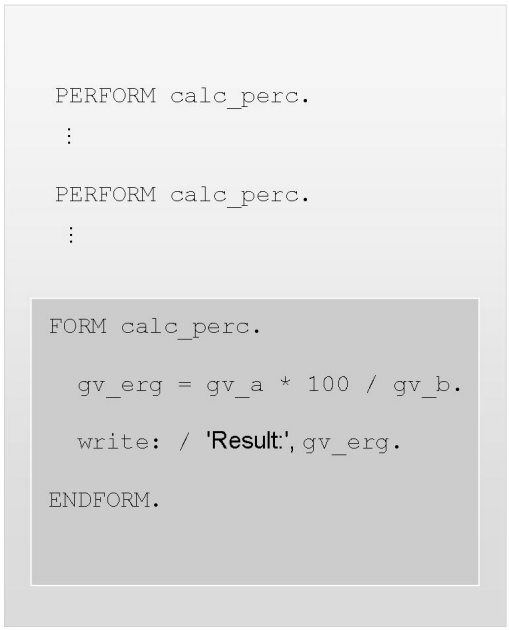


Figure 96: Simple Example of a Subroutine

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

Parameter Definition for Subroutines

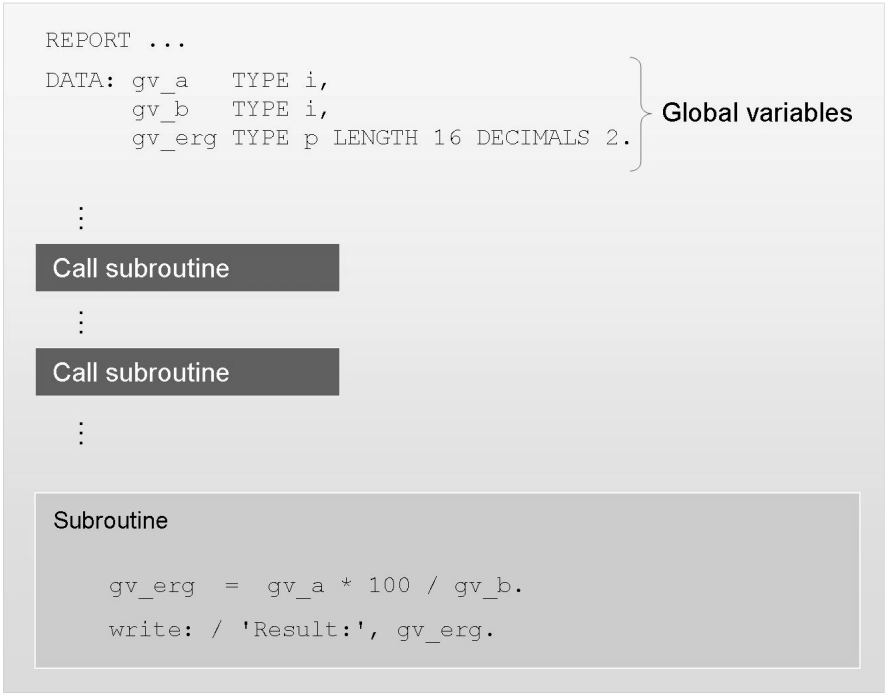


Figure 97: Passing Parameters - Visibility of Global Variables

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

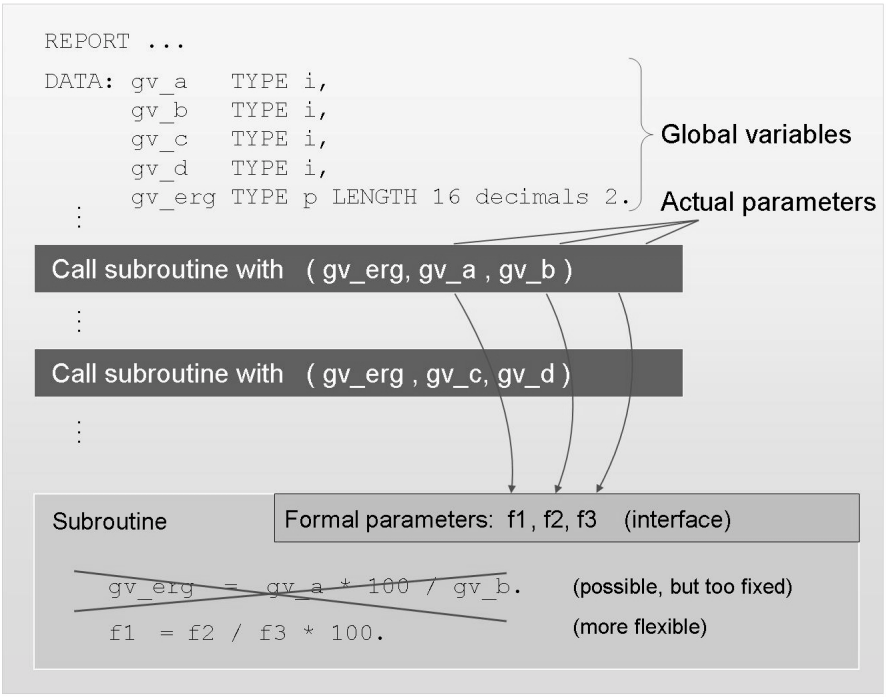


Figure 98: Passing Parameters - Defining an Interface

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

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

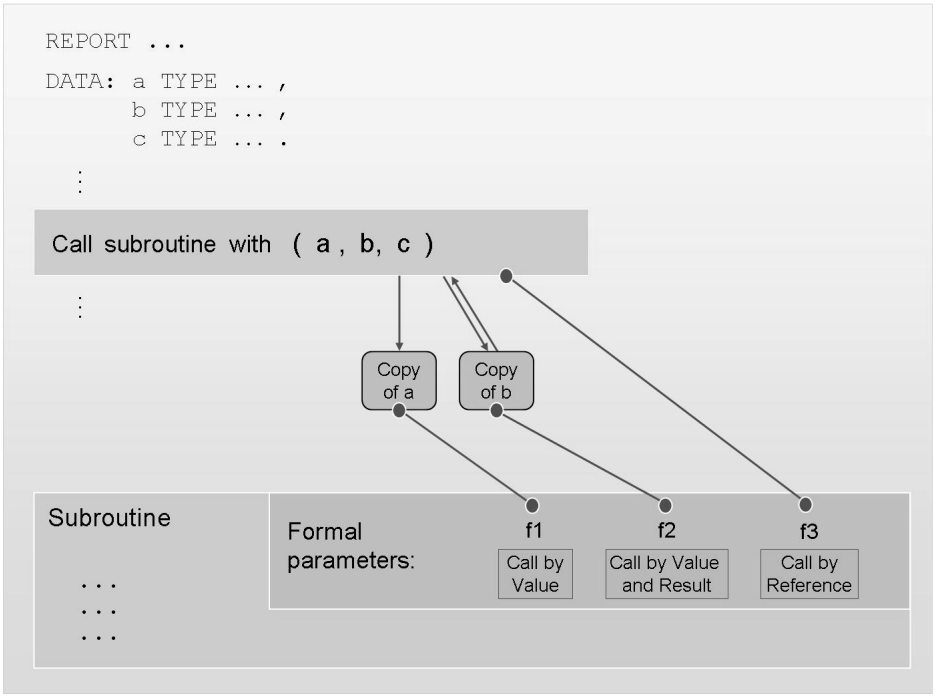


Figure 99: Ways of Passing Interface Parameters

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

There are three pass types for subroutines:

Call by Value

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

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

Call by Value and Result

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

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

Call by Reference

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

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

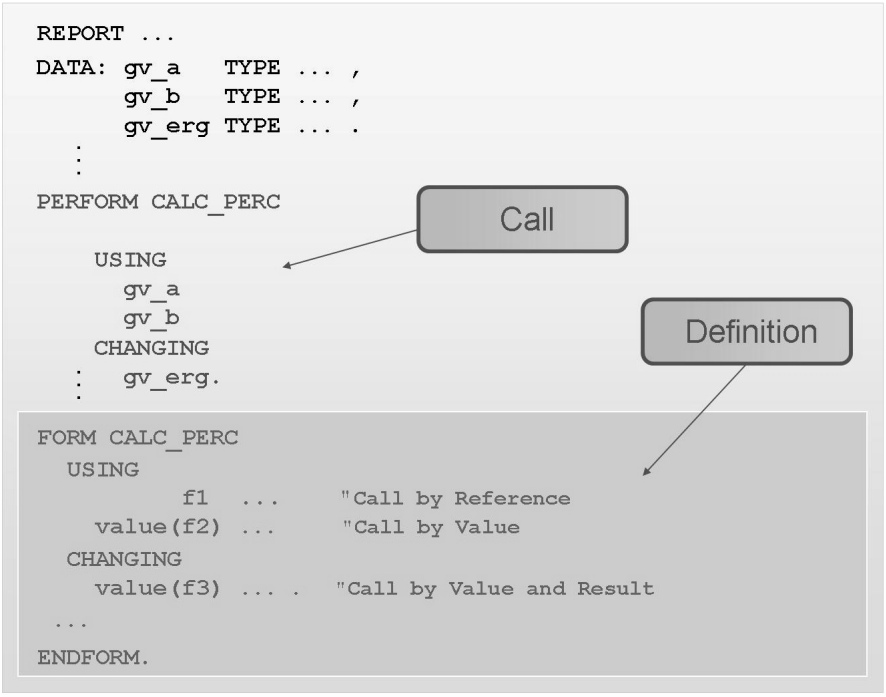


Figure 100: Defining and Calling Subroutines

Structure of a Subroutine

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

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

Call by Value

You list each of the formal parameters that should have the pass type "Call by Value" (here: *f1*) with the **`VALUE` prefix** under **`USING`**. (Refer to the above graphic for the syntax.)

Call by Value and Result

You list each of the formal parameters that should have the pass type "Call by Value and Result" (here: *f2*) with the **`VALUE` prefix** under **`CHANGING`**. (Refer to the above graphic for the syntax.)

Call by Reference

You list each of the formal parameters that should have the pass type "Call by Reference" (here: *f3*) **without the** `VALUE` prefix under **`CHANGING`**.(Refer to the above graphic for the syntax.)

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

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

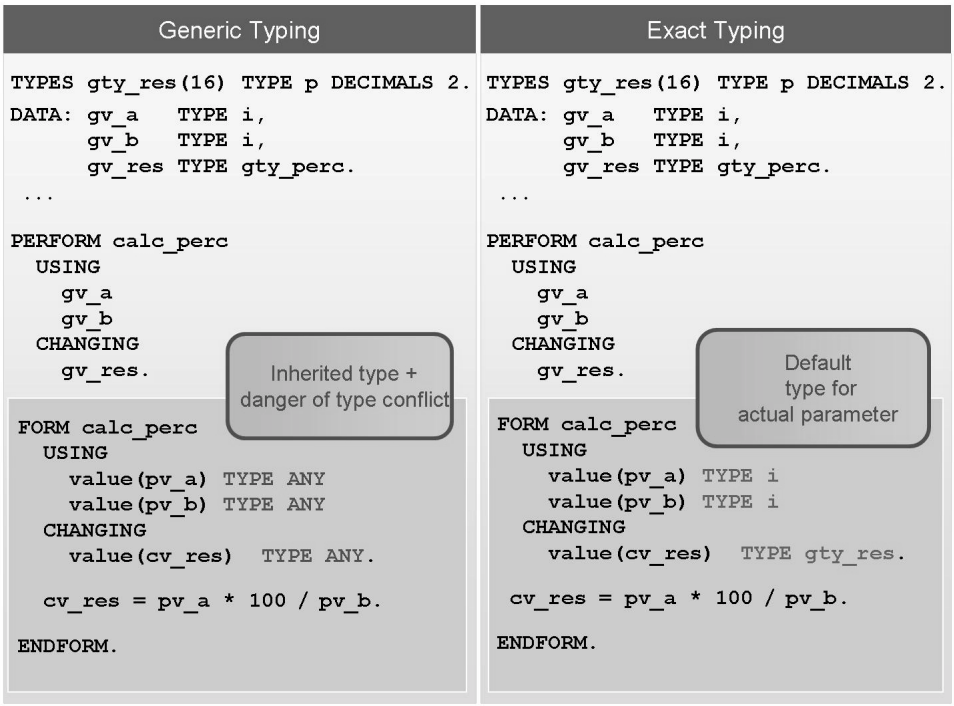


Figure 101: Typing the Interface Parameters

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

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

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

Local and Global Data Objects

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

DATA: gv_a      TYPE i,
      gv_b      TYPE i,
      gv_result TYPE gty_perc.
...
PERFORM calc_perc
           USING gv_a
                 gv_b
        CHANGING gv_result.
...
```

Global variables

```
FORM calc_perc
  USING
    pv_a TYPE i
    pv_b TYPE i
  CHANGING
    cv_res TYPE gty_perc.

  DATA lv_pc TYPE p LENGTH 16 DECIMALS 1.

...

ENDFORM.
```

Formal parameters (only visible locally)

Local variables (only visible locally)

Figure 102: Visibility of Global and Local Data Objects

Exercise 14: Subroutines

Exercise Objectives

After completing this exercise, you will be able to:

- Create subroutines
- Use the subroutine interface to pass data

Business Example

Change your program for calculating basic arithmetic operations so that you calculate the power from a base number and an exponent. Implement this calculation in a subroutine.

Template:

BC100_MOT_SUBROUTINE

Solution:

BC100_MOS_SUBROUTINE

Task 1:

Copy the program BC100_MOT_SUBROUTINE to the name **ZBC100_##_SUBROUTINE**.

1. Copy the copy template.

Task 2:

Create a subroutine (suggested name: **CALC_POWER**) that carries out an exponential calculation using two input parameters and returns the result. This subroutine calculates the power of an integer.

1. At the end of your program, create a subroutine for exponential calculation.
2. Define two USING parameters for transferring the operands (suggested name: **PV_INT1** and **PV_INT2**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.

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

Continued on next page

3. Implement the exponential calculation in the subroutine.

Task 3:

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

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

Hint: Generate the subroutine call by choosing the *Pattern* pushbutton.

Task 4:

Test and debug your program.

1. Execute your program and check the result.
2. Follow the program flow with the Debugger.

Solution 14: Subroutines

Task 1:

Copy the program BC100_MOT_SUBROUTINE to the name **ZBC100_##_SUBROUTINE**.

1. Copy the copy template.
   a) Carry out this exercise as usual using transaction SE38 (ABAP Editor).

Task 2:

Create a subroutine (suggested name: **CALC_POWER**) that carries out an exponential calculation using two input parameters and returns the result. This subroutine calculates the power of an integer.

1. At the end of your program, create a subroutine for exponential calculation.
   a) See the source code excerpt from the model solution.
2. Define two USING parameters for transferring the operands (suggested name: **PV_INT1** and **PV_INT2**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.

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

   a) See the source code excerpt from the model solution.
3. Implement the exponential calculation in the subroutine.
   a) See the source code excerpt from the model solution.

Task 3:

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

1. Extend the IF or CASE structure with a branch that is processed if the parameter contains the value “P”.
   a) See the source code excerpt from the model solution.

Continued on next page

2. In this branch, call the subroutine and supply the parameters with suitable actual parameters.

Hint: Generate the subroutine call by choosing the *Pattern* pushbutton.

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

Task 4:

Test and debug your program.

1. Execute your program and check the result.
   a) Carry out this step in the usual way.
2. Follow the program flow with the Debugger.
   a) Perform this step as described in the course materials.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_MOS_SUBROUTINE
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_mos_subroutine.

TYPES gty_result     TYPE p LENGTH 16 DECIMALS 2.

PARAMETER: pa_int1   TYPE i DEFAULT 4,
           pa_op     TYPE c DEFAULT 'P',
           pa_int2   TYPE i DEFAULT 3.

DATA:      gv_result TYPE gty_result.

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

Continued on next page

Internal Use SAP Partner Only

```
  WHEN '/'.
    IF pa_int2 = 0.
     WRITE:  / 'Incorrect operator!'.
    ELSE.
      gv_result = pa_int1 / pa_int2.
    ENDIF.
  WHEN '%'.
    IF pa_int2 = 0.
      WRITE:  / 'Incorrect operator!'.
    ELSE.
      gv_result = pa_int1 * 100 / pa_int2.
    ENDIF.
  WHEN 'P'.
    PERFORM calc_power
                  USING
                    pa_int1
                    pa_int2
                  CHANGING
                    gv_result.
  WHEN OTHERS.
    WRITE: / 'Falscher Operator!'.
ENDCASE.

write: / 'Result:', gv_result.
*&---------------------------------------------------------------------*
*&      Form  POWER
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*      -->P_PA_INT1  text
*      -->P_PA_INT2  text
*      <--P_GV_RESULT  text
*----------------------------------------------------------------------*
FORM calc_power  USING    pv_int1   TYPE i
                          pv_int2   TYPE i
                 CHANGING cv_result TYPE gty_result.

  IF pv_int2 = 0.
    cv_result = 1.
  ELSE.
    cv_result = pv_int1.

    WHILE sy-index < pv_int2.
      cv_result =  cv_result * pv_int1.
    ENDWHILE.
```

Continued on next page


```
  ENDIF.

ENDFORM.                    " POWER
```

Lesson Summary

You should now be able to:

- Develop simple subroutines
- Pass parameters to these subroutines

Lesson: Exiting Subroutines

Lesson Overview

In this lesson, you learn how subroutines can be exited prematurely.

Lesson Objectives

After completing this lesson, you will be able to:

- Exit subroutines prematurely

Business Example

As an employee of "Calculate & Smile", you want to learn how to exit a subroutine prematurely in situations when the entire subroutine does not have to be run through, or when the program would terminate if the subroutine is not exited.

Exiting the Subroutine Without Return of Parameters

If parameters are passed to a subroutine with “Call by Value and Result” and this subroutine is terminated prematurely by the `STOP` statement or a type E user message, the returning of a parameter is suppressed.

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

DATA: gv_a   TYPE i,
      gv_b   TYPE i,
      gv_res TYPE gty_perc.
...
PERFORM calc_perc
           USING gv_a
                 gv_b
        CHANGING gv_res.
...
```

```
FORM calc_perc
  USING
    pv_a TYPE i
    pv_b TYPE i
  CHANGING
    cv_res TYPE gty_perc.

   IF pv_b = 0.
      WRITE /  'Error in percentage calculation! '(epc) .
      STOP.
   ENDIF.
   cv_res = pv_a * 100 / pv_b.
ENDFORM.
```

Figure 103: Exiting Subroutines with the STOP Statement

The subroutine is exited immediately with the STOP statement and the program is exited.

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

DATA: gv_a   TYPE i,
      gv_b   TYPE i,
      gv_res TYPE gty_perc.
...
PERFORM calc_perc
           USING gv_a
                 gv_b
        CHANGING gv_res.
...
```

```
FORM calc_perc
  USING
    pv_a TYPE i
    pv_b TYPE i
  CHANGING
    cv_res  TYPE gty_perc.

   If pv_b = 0.
      MESSAGE E000(BC100).
*     Error in percentage calculation
   endif.
   cv_res = pv_a * 100 / pv_b.
ENDFORM.
```

Figure 104: Subroutine Termination with an Error Message

The error MESSAGE statement interrupts the program flow and displays the short text of a message in the logon language of the current user.

Exiting the Subroutine with Return of Parameters

If parameters are passed to a subroutine with “Call by Value and Result” and this subroutine is exited prematurely with the RETURN statement, the parameters are returned.

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

DATA: gv_a   TYPE i,
      gv_b   TYPE i,
      gv_res TYPE gty_perc.
...
PERFORM calc_perc
           USING gv_a
                 gv_b
        CHANGING gv_res.
WRITE: / 'Result', gv_res.
```

```
FORM calc_perc
  USING
    pv_a TYPE i
    pv_b TYPE i
  CHANGING
    cv_res  TYPE gty_perc.

   IF pv_b = 0.
      WRITE /  'Error in percentage calculation! '(epc) .
      RETURN.
   ENDIF.
   cv_res = pv_a * 100 / pv_b.
ENDFORM.
```

Figure 105: Exiting the Subroutine with RETURN

The RETURN statement ends the current processing block immediately. It can be inserted anywhere in the processing block. The list processor for displaying the basic list is then called.

Hint: We recommend that you use the CHECK and EXIT statements only within loops. You should use RETURN to exit subroutines.

Exercise 15: Exiting Subroutines

Exercise Objectives

After completing this exercise, you will be able to:

- Exit a subroutine prematurely

Business Example

Modify your calculator program for the basic arithmetic operations so that you can calculate the power also in a subroutine. Exit the subroutine with an “e” message if division by zero occurs.

Template:

BC100_MOT_SUBROUTINE_RETURN

Solution:

BC100_MOS_SUBROUTINE_RETURN

Task 1:

Copy the program BC100_MOT_SUBROUTINE_RETURN to the name **ZBC100_##_SUBROUTINE_RETURN**.

1. Copy the copy template.

Task 2:

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

1. At the end of your program, create a subroutine for the percentage calculation.
2. Define two USING parameters for transferring the operands (suggested name: **PV_INT1** and **PV_INT2**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.

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

Continued on next page

3. Implement the power calculation in the subroutine.

Task 3:

Extend the subroutine so that it is exited if the program divides by zero.

1. If the parameter **pv_int2** is zero, the subroutine is to exit with an “e” message. Use the sample statement to do this. Use the message class **BC100**, the type **E**, and the number **000**.

Task 4:

Test and debug your program.

1. Execute your program and check the result.
2. Follow the program flow with the Debugger.

Solution 15: Exiting Subroutines

Task 1:

Copy the program BC100_MOT_SUBROUTINE_RETURN to the name **ZBC100_##_SUBROUTINE_RETURN**.

1. Copy the copy template.

   a) Carry out this exercise as usual using transaction SE38 (ABAP Editor).

Task 2:

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

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

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

2. Define two USING parameters for transferring the operands (suggested name: **PV_INT1** and **PV_INT2**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.

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

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

3. Implement the power calculation in the subroutine.

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

Task 3:

Extend the subroutine so that it is exited if the program divides by zero.

1. If the parameter **pv_int2** is zero, the subroutine is to exit with an “e” message. Use the sample statement to do this. Use the message class **BC100**, the type **E**, and the number **000**.

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

Continued on next page

Task 4:

Test and debug your program.

1. Execute your program and check the result.
   a) Carry out this step in the usual way.
2. Follow the program flow with the Debugger.
   a) Perform this step as described in the course materials.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_MOS_SUBROUTINE_RETURN
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_mos_subroutine_return.

TYPES gty_result     TYPE p LENGTH 16 DECIMALS 2.

PARAMETER: pa_int1   TYPE i DEFAULT 6,
           pa_op     TYPE c DEFAULT '%',
           pa_int2   TYPE i DEFAULT 0.

DATA:      gv_result TYPE gty_result.

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    IF pa_int2 = 0.
      WRITE: / 'Division durch Null!'.
    ELSE.
      gv_result = pa_int1 / pa_int2.
    ENDIF.
  WHEN '%'.
    PERFORM calc_percentage
                 USING
```

Continued on next page


```
                    pa_int1
                    pa_int2
                  CHANGING
                    gv_result.
  WHEN 'P'.
    PERFORM calc_power
                  USING
                    pa_int1
                    pa_int2
                  CHANGING
                    gv_result.
  WHEN OTHERS.
    WRITE: / 'Falscher Operator!'.
ENDCASE.

WRITE: / 'Ergebnis:', gv_result.
*&---------------------------------------------------------------------*
*&      Form  POWER
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*      -->P_PA_INT1  text
*      -->P_PA_INT2  text
*      <--P_GV_RESULT  text
*----------------------------------------------------------------------*
FORM calc_power  USING    pv_int1   TYPE i
                          pv_int2   TYPE i
                 CHANGING cv_result TYPE gty_result.

  IF pv_int2 = 0.
    cv_result = 1.
  ELSE.
    cv_result = pv_int1.

    WHILE sy-index < pv_int2.
      cv_result =  cv_result * pv_int1.
    ENDWHILE.
  ENDIF.

ENDFORM.                    " POWER
*&---------------------------------------------------------------------*
*&      Form  CALC_PERCENTAGE
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
```

Continued on next page


```
*      -->P_PA_INT1  text
*      -->P_PA_INT2  text
*      <--P_GV_RESULT  text
*---------------------------------------------------------------------*
FORM calc_percentage  USING    pv_int1   TYPE i
                               pv_int2   TYPE i
                      CHANGING cv_result TYPE gty_result.

  IF pv_int2 = 0.
    MESSAGE e000(zbc100).
*   Division durch Null!

  ELSE.
    cv_result = pv_int1 * 100 / pv_int2.
  ENDIF.

ENDFORM.                    " CALC_PERCENTAGE
```

Lesson Summary

You should now be able to:

- Exit subroutines prematurely

Lesson: Debugging Subroutines

Lesson Overview

In this lesson, you learn how to execute subroutines in debugging mode and how the debugging of a subroutine can be ended prematurely.

Lesson Objectives

After completing this lesson, you will be able to:

- Debug subroutines

Business Example

As an employee of "Calculate & Smile", you want to activate the Debugger at runtime to execute subroutines in debugging mode. You need to learn how to execute a subroutine without debugging the source code and how a subroutine can be exited prematurely in debugging mode. The Debugger makes it easier for you to understand new programs and it helps you find errors.

Subroutines in the Debugger

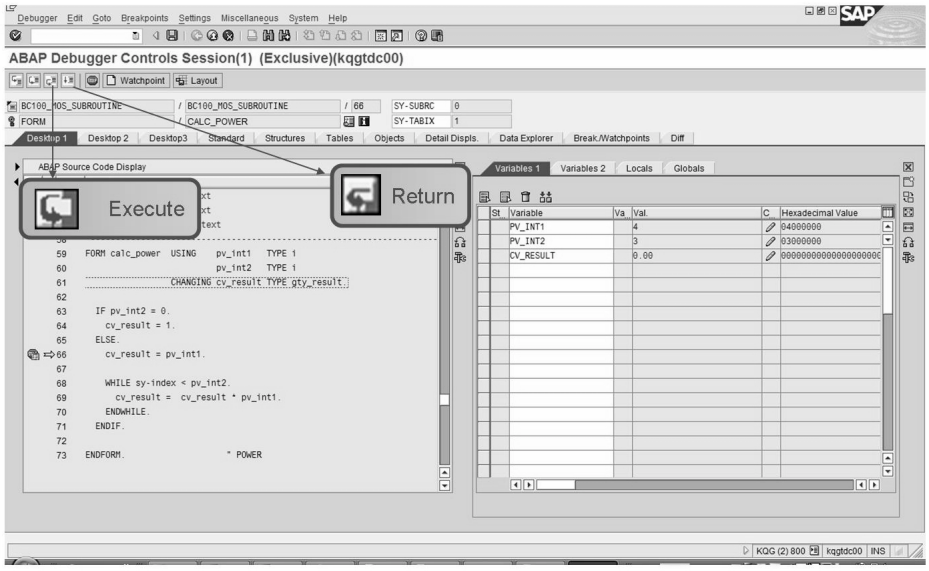


Figure 106: Subroutines in Debugging Mode

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

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

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

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

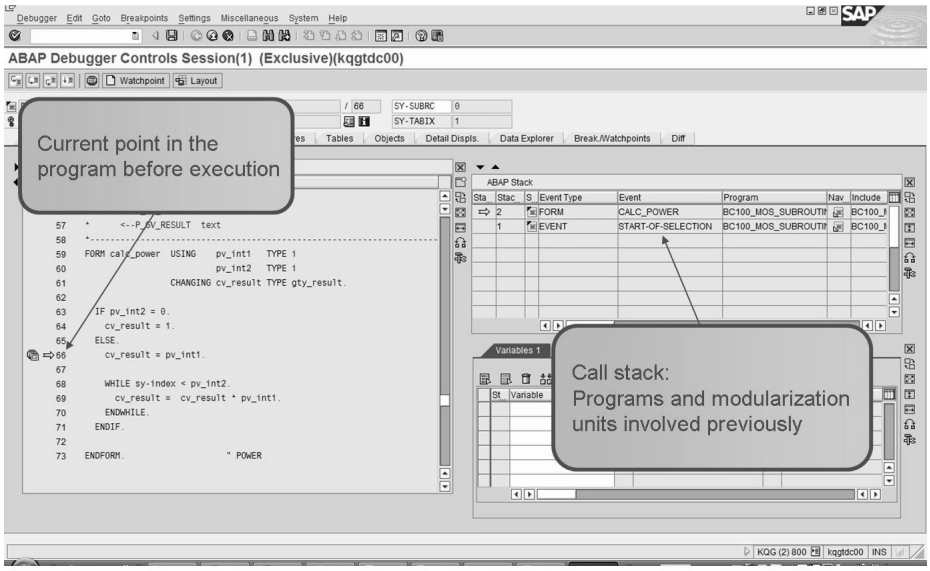


Figure 107: Debugging - Call Stack

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

Exercise 16: Debugging Subroutines

Exercise Objectives

After completing this exercise, you will be able to:

- Debug subroutines

Business Example

Debug your program for calculating the basic arithmetic operations. Change data objects and use the Debugger functions “RETURN” and “EXECUTE”.

Template:

BC100_MOS_SUBROUTINE_RETURN2

Task 1:

Copy the program BC100_MOS_SUBROUTINE_RETURN2 to the name **ZBC100_##_SUBROUTINE_RETURN2**.

1. Copy the copy template.

Task 2:

Test and debug your program.

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

Solution 16: Debugging Subroutines

Task 1:

Copy the program BC100_MOS_SUBROUTINE_RETURN2 to the name **ZBC100_##_SUBROUTINE_RETURN2**.

1. Copy the copy template.
   a) Carry out this exercise as usual using transaction SE38 (ABAP Editor).

Task 2:

Test and debug your program.

1. Execute your program and check the result.
   a) Carry out this step in the usual way.
2. Follow the program flow in the Debugger. Use the function keys or pushbuttons to exit or skip the subroutine.
   a) Carry out this step in the usual way.

Result

```
*&---------------------------------------------------------------------*
*& Report  BC100_MOS_SUBROUTINE_RETURN2
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  bc100_mos_subroutine_return2.

TYPES gty_result     TYPE p LENGTH 16 DECIMALS 2.

PARAMETER: pa_int1   TYPE i DEFAULT 6,
           pa_op     TYPE c DEFAULT '%',
           pa_int2   TYPE i DEFAULT 0.

DATA:      gv_result TYPE gty_result.

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
```

Continued on next page

Internal Use SAP Partner Only

```
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    IF pa_int2 = 0.
      WRITE: / 'Division durch Null!'.
    ELSE.
      gv_result = pa_int1 / pa_int2.
    ENDIF.
  WHEN '%'.
    PERFORM calc_percentage
                  USING
                    pa_int1
                    pa_int2
                  CHANGING
                    gv_result.
  WHEN 'P'.
    PERFORM calc_power
                  USING
                    pa_int1
                    pa_int2
                  CHANGING
                    gv_result.
  WHEN OTHERS.
    WRITE: / 'Falscher Operator!'.
ENDCASE.

WRITE: / 'Ergebnis:', gv_result.
*&---------------------------------------------------------------------*
*&      Form  POWER
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*      -->P_PA_INT1  text
*      -->P_PA_INT2  text
*      <--P_GV_RESULT  text
*----------------------------------------------------------------------*
FORM calc_power  USING    pv_int1   TYPE i
                          pv_int2   TYPE i
                 CHANGING cv_result TYPE gty_result.

  IF pv_int2 = 0.
    cv_result = 1.
  ELSE.
    cv_result = pv_int1.
```

Continued on next page


```
    WHILE sy-index < pv_int2.
      cv_result =  cv_result * pv_int1.
    ENDWHILE.
  ENDIF.

ENDFORM.                    " POWER
*&---------------------------------------------------------------------*
*&      Form  CALC_PERCENTAGE
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*      -->P_PA_INT1  text
*      -->P_PA_INT2  text
*      <--P_GV_RESULT  text
*----------------------------------------------------------------------*
FORM calc_percentage  USING    pv_int1   TYPE i
                               pv_int2   TYPE i
                      CHANGING cv_result TYPE gty_result.

  IF pv_int2 = 0.
    WRITE: / 'Division durch Null!'.
    RETURN.
  ELSE.
    cv_result = pv_int1 * 100 / pv_int2.
  ENDIF.

ENDFORM.                    " CALC_PERCENTAGE
```

Lesson Summary

You should now be able to:

- Debug subroutines

Internal Use SAP Partner Only

Unit Summary

You should now be able to:

- Develop simple subroutines
- Pass parameters to these subroutines
- Exit subroutines prematurely
- Debug subroutines

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Course Summary

You should now be able to:

- Describe the basic concepts of programming and implement what you have learned in simple ABAP programs

Index

Internal Use SAP Partner Only

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.

Internal Use SAP Partner Only