BC100

Introduction to Programming with ABAP

PARTICIPANT HANDBOOK
INSTRUCTOR-LED TRAINING

Course Version: 18
Course Duration: 2 Day(s)
Material Number: 50146852

SAP Copyrights, Trademarks and Disclaimers

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation	
Demonstration	
Procedure	
Warning or Caution	
Hint	
Related or Additional Information	
Facilitated Discussion	
User interface control	*Example text*
Window title	*Example text*

Contents

Course Overview

TARGET AUDIENCE
This course is intended for the following audiences:

- Developer
- Development Consultant
- IT Support

UNIT 1 Basics of ABAP Programming

UNIT OBJECTIVES

- Develop a simple ABAP program
- Describe ABAP syntax
- Add comments to code
- Consult keyword documentation
- Implement a simple dialog using the PARAMETERS statement
- Customize the ABAP Editor

Unit 1
Lesson 1

Developing a Simple ABAP Program

LESSON OVERVIEW
This lesson introduces you to the ABAP Editor and shows you how to develop a simple ABAP program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Develop a simple ABAP program

Basics of ABAP Programs and the ABAP Editor

The best way to learn how to program in ABAP (or any programming language) is to start writing code. This course takes that approach.

To start, we will explore the development environment in which you will work. The tool that ABAP developers work with most is called the ABAP Editor. You can access the ABAP Editor by using transaction code `SE38`.

To create and run your first ABAP program, you need to understand the following:

- How to create a new program
- How to work with the Editor
- The most important Editor functions
- How to formulate a correct ABAP statement

Creating Programs in the ABAP Editor

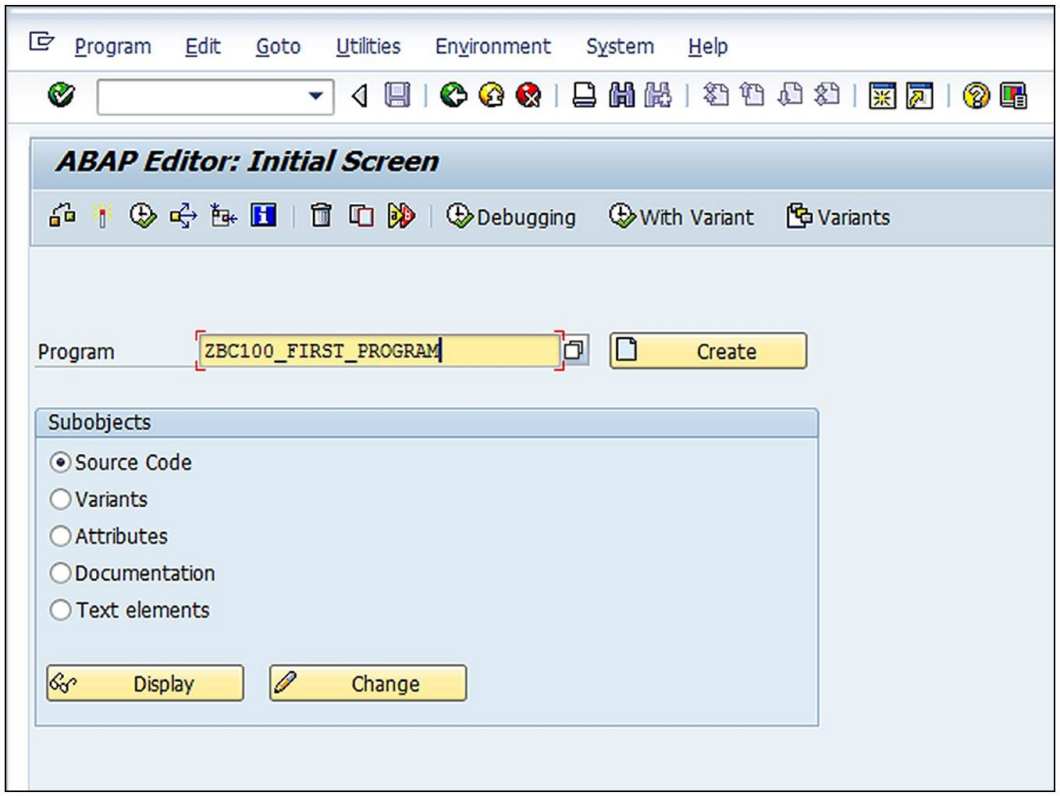

Figure 1: Creating Programs in the ABAP Editor

To create a new program in the ABAP Editor, you must first enter a name. For custom programs, you must use the customer namespace (that is, use a specific naming convention). Custom program names must begin with **Z** or **Y**, as shown in the figure Creating Programs in the ABAP Editor.

Program Attributes

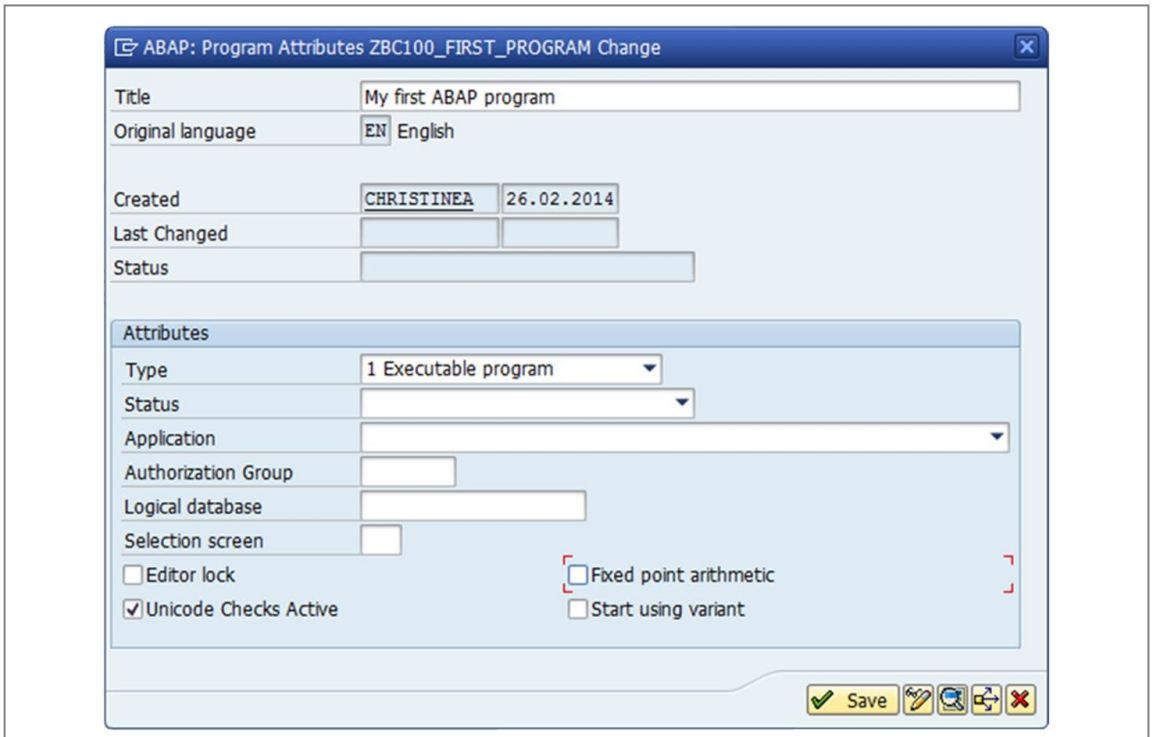

Figure 2: Program Attributes

When you create a new program, you must set some important program attributes such as the program title (which typically differs from the technical program name) and the program type. This course focuses on executable programs, but there are other program types also. Executable programs can be used, for example, when you require a simple ABAP report.

After you assign the program attributes, choose whether to assign the program to a package or create it as a local object. Packages are used to categorize ABAP programs and other ABAP objects, and are necessary if you want to transport the program to further systems in the landscape. After you assign a program to a package, the system asks you to assign it to a transport request. If you create a program as a local object, the program is not transportable and you are not prompted for the transport request number.

Note:
Other program types and packages are covered in more detail in the course ABAP Workbench Foundations (BC400).

The ABAP Editor

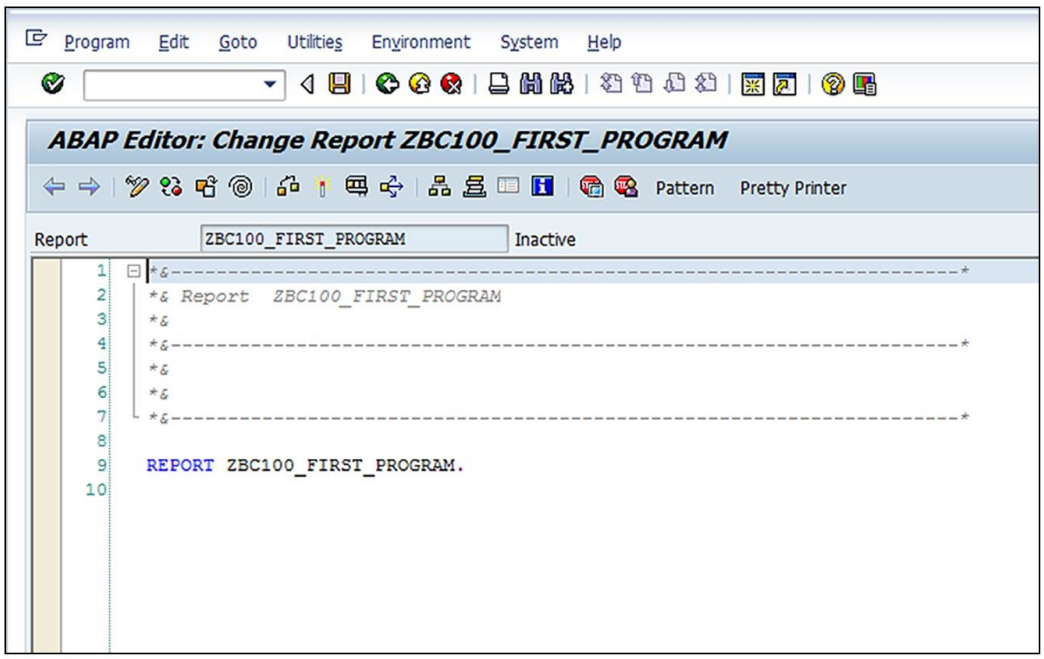

Figure 3: The ABAP Editor

After you assign the program to a package (or create it as a local object), the ABAP Editor opens. You can now begin writing code.

In this course, you will learn the main functionality of the ABAP Editor and the correct syntax to use when writing ABAP code. Syntax is important: just as you need to learn the correct words to communicate with someone in a different language, you also need to learn the correct language to communicate in an ABAP program. You will also learn how to navigate the user interface and find the options you need.

Note:
If your Editor does not resemble the interface shown in the figure, The ABAP Editor, you might be using an older version with fewer features. Check if the new Editor is available in your system by choosing *Utilities* → *Settings*, then choosing the *ABAP Editor* → *Editor* tab page. If you see the *Front-End Editor (New)* radio button, select it.

Basic ABAP Syntax Rules

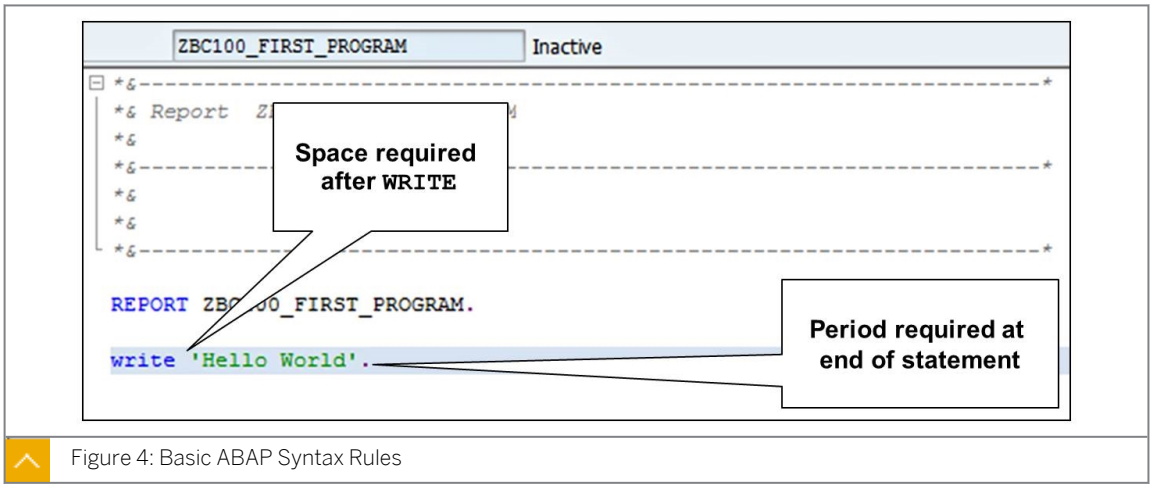

Figure 4: Basic ABAP Syntax Rules

Characteristics of ABAP Syntax

- ABAP programs consist of individual statements.
- The first word in a statement is called an ABAP keyword.
- Each statement ends with a period.
- Two words must be separated by a space.
- Statements can be indented.
- The ABAP runtime system does not differentiate between upper and lowercase in keywords, additions, and operands.

Key Functions in the ABAP Editor

Table 1: Key Functions in the ABAP Editor

Button	Keyboard Command	Description
	**CTRL** + **S**	*Save*
	**CTRL** + **F2**	*Check Syntax*
	CTRL + **F3**	*Activate*
	**F8**	*Direct Processing* (to test or execute the program)

There are four key functions that you must know when writing code in the ABAP Editor.

- *Save*

  Saves your code, but does not check the syntax for correctness or activate the program. It is good practice to save your work regularly.

- *Check Syntax*

Checks that all syntax in your program is correct. For example, if you have misspelled the keyword `WRITE` or forgotten to end a statement with a period, the syntax check will return a message that indicates the problem.

- *Activate*

  Activates the program. When you save a program, the system stores an inactive version of the program in the Repository. Activating a program generates the runtime object that is used when the program is executed. It is important to activate your programs when you finish developing them: a transport request can be released only if the programs inside it are active, and a transaction code linked to an ABAP program will run the active version of that program.

- *Direct Processing*

  Executes the program. As a developer, you need to know whether your code behaves as expected. Use this function to test your program regularly.

LESSON SUMMARY

You should now be able to:

- Develop a simple ABAP program

Unit 1
Lesson 2

Introducing ABAP Syntax

LESSON OVERVIEW
In this lesson, you learn the basics of ABAP syntax, how to add comments to code, and how to consult the documentation for ABAP keywords.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Describe ABAP syntax
- Add comments to code
- Consult keyword documentation

Characteristics of the ABAP Language

Advanced Business Application Programming (ABAP) is a programming language developed by SAP for programming business applications in the SAP environment. The following list outlines some of the key characteristics of ABAP.

Key Characteristics of ABAP

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

ABAP is especially designed for dialog-based business applications. To support the type-specific processing of data, different numeric and character-like types are available.

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

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

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

ABAP syntax has the same meaning or function, regardless of the relational database system and operating system for the application and presentation server.

Applications implemented in ABAP will run in future releases.

General ABAP Syntax

ABAP programs are comprised of individual sentences (statements). The following list describes the most important things to remember about ABAP statements.

ABAP Statements

- Start with a keyword
- End with a period (without exception)
- May contain additions and operands (depending on the keyword used)
- Can span multiple lines
- Words must be separated by at least one space

You can indent statements to enhance the readability of the code. You can also use upper or lowercase: with keywords, additions, and operands, the ABAP runtime system does not differentiate between cases. You can use a tool called the Pretty Printer to help you indent your code and convert code between uppercase and lowercase. The Pretty Printer is discussed in the lesson Customizing the ABAP Editor.

Hint:
Although the ABAP runtime system does not differentiate between upper and lowercase, it is customary to write keywords and their additions in uppercase letters and operands in lowercase. This convention is used in this course.

Example Statements

```
* Some ABAP statements consist ONLY of a keyword

NEW-LINE.

* Other ABAP statements can be very long

SELECT SINGLE carrid carrname currcode
  FROM scarr
    INTO CORRESPONDING FIELDS OF gs_carrier
      WHERE carrid = 'AA'.
```

Figure 5: What Goes Between the Keyword and the Period?

This example code demonstrates two ABAP statements. Some statements consist of a keyword only, such as the `NEW-LINE` statement in this code block. However, there are many ABAP keywords where you must use a much longer statement (such as the `SELECT` statement in this code block).

This course introduces many keywords and explains their use. It will also explain how to access the complete documentation for every ABAP keyword, which you can refer to when developing your own programs.

Chained Statements

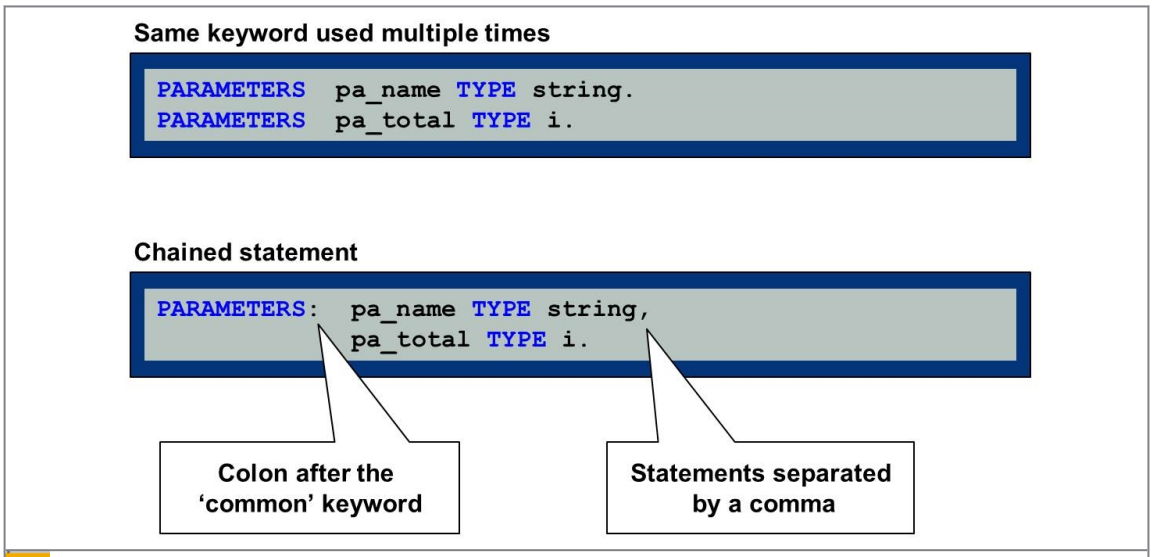

Figure 6: Chained Statements

The figure Chained Statements shows two examples. In the first example, the two statements use the same keyword. To avoid repeating the keyword, you can link the two statements in a chained statement.

Chained statements allow you to combine **consecutive statements** with an **identical keyword**. To create a chained statement, follow these steps:

1. Write the identical beginning part of the statements (the keyword), followed by a colon.
2. 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).

Note:
Chained statements do not improve performance. They merely represent a simplified form of syntax.

Comments

```
PARAMETERS: pa_name TYPE string,      "Input field for users name
            pa_total TYPE i,          "Total days annual vacation
            pa_used TYPE i,           "Number of days vacation already used
            pa_roll TYPE i.           "Number of days vacation to roll over to next year

DATA gv_remaining TYPE i.             "Variable to hold remaining days vacation for the year
DATA gv_next_year TYPE i.             "Variable to hold number of days vacation for next year

* Calculate the remaining days vacation for the year

gv_remaining = pa_total - pa_used.

* Output the remaining days vacation for the year

WRITE: gv_remaining, 'days vacation remaining for'(rem), pa_name.

* Calulate the number of days vacation for next year

gv_next_year = pa_total + pa_roll.
```

Figure 7: Comments

A comment is an explanation that is added to a program to help others understand the code. Comments are ignored when the program is generated. They are not executable statements.

There are two different types of comments in ABAP:

- The * character at the start of a program line indicates that the entire line is a comment. Therefore, the ABAP runtime system ignores the whole line.
- The " character, which can be entered at any position in the line, indicates that the remaining content in the line is a comment.

It is good practice to place whole line comments on the line above the statements that they describe. End of line comments are most suitable for declarations, rather than executable statements. The figure Comments shows both types of comments.

Keyword Documentation

```
* Split the full name entered into first name and surname

SPLIT pa_name AT ' ' INTO gv_first_name gv_last_name.

* Calculate the remaining days holiday for the year

CALL FUNCTION 'BC100_CALC_HOLS'
  EXPORTING
    iv_total           = pa_total
    iv_used            = pa_used
 IMPORTING
   EV_REMAINING       = gv_remaining.
```

What additions does the SPLIT statement have?

What does the CALL FUNCTION statement do?

Figure 8: Learning More About ABAP Keywords

Even experienced ABAP developers sometimes encounter syntax with which they are not familiar, or need to find information about a certain ABAP topic quickly. The ABAP Editor includes keyword documentation for all syntax so that you can find what you need.

ABAP Keyword Documentation

There are two ways to open this documentation:

- Place your cursor on the keyword that you want to learn about, then press F1.

  This opens the documentation for that specific statement.

- Choose the (*Help On*) button.

  This displays a dialog box in which you can enter the required keyword or browse for more general information (as shown in the figure ABAP Keyword Documentation). For example, there is a glossary of keywords, an ABAP Subject Directory, and other useful resources.

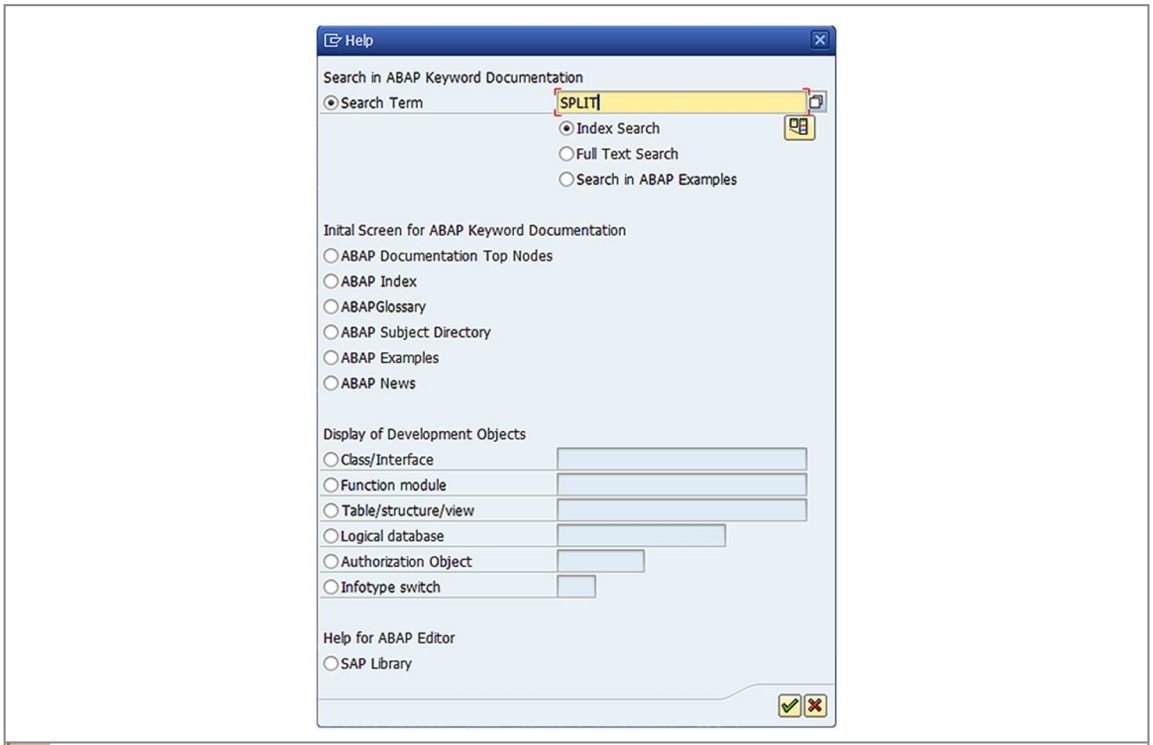

Figure 9: ABAP Keyword Documentation

The ABAP Keyword Documentation is an important tool for all ABAP developers. Explore the documentation to familiarize yourself with the contents and learn how to quickly research unfamiliar topics.

LESSON SUMMARY

You should now be able to:

- Describe ABAP syntax
- Add comments to code
- Consult keyword documentation

Implementing a Simple Dialog

LESSON OVERVIEW
This lesson shows you how to use the `PARAMETERS` statement to create a simple dialog in an ABAP program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Implement a simple dialog using the PARAMETERS statement

Simple Dialog with the PARAMETERS Statement

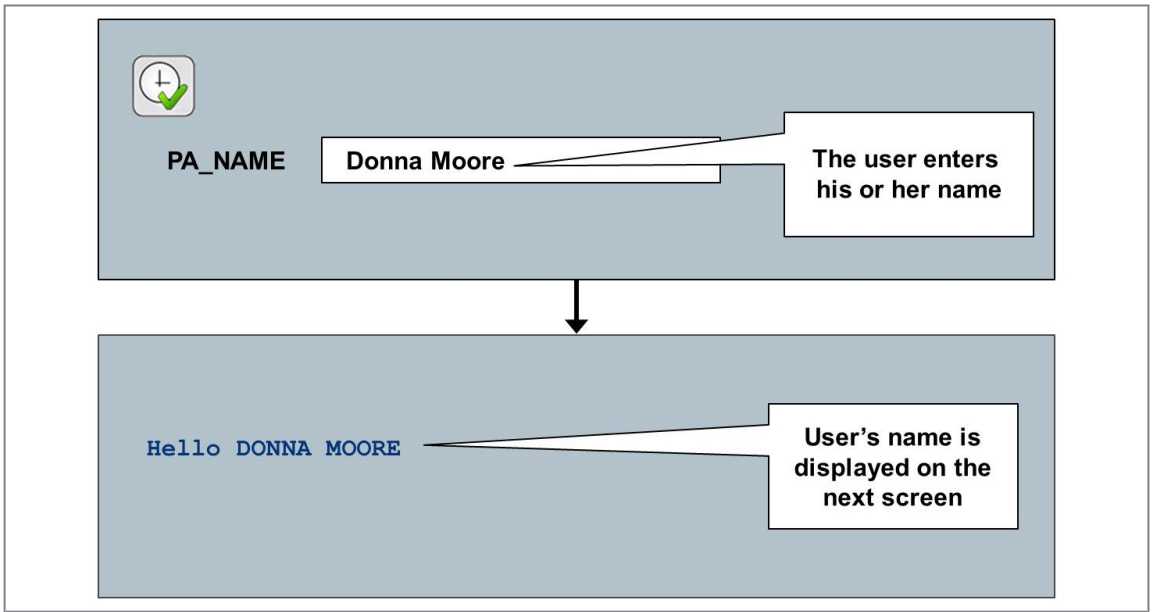

Figure 10: Hello World, or Something More Personal?

The `PARAMETERS` keyword defines an input field on what is known as a **Selection Screen**. Every parameter you define in your ABAP program has a single corresponding input field on the selection screen. The selection screen also includes an *Execute* button that the user chooses once he or she fills the supplied input field (or fields).

Note:
Although there is a tool called the Screen Painter that is used to design screens, this tool is not used here. Instead, the selection screen is generated based on the parameter declarations in your ABAP code.

The Parameters Keyword

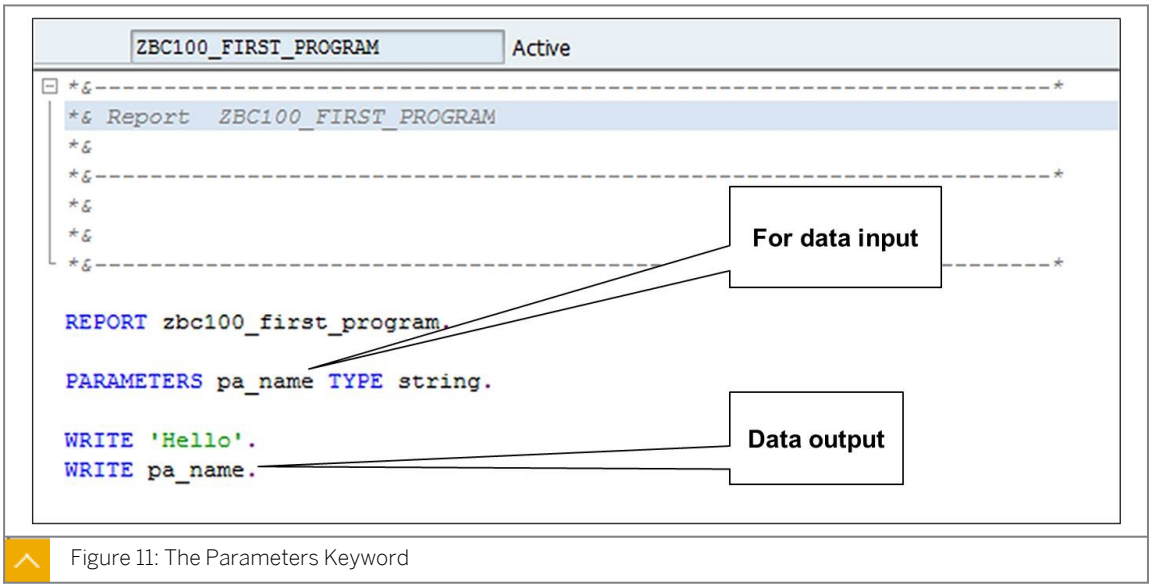

Figure 11: The Parameters Keyword

The figure, The Parameters Keyword, shows an example of a parameter declaration. Parameters (such as `pa_name` in the figure) can also be thought of as placeholders: you want to output the name entered by the user, but you cannot know what the user's name is while you are developing the program. At program execution, the user can enter their name.

When you define the parameter, the program reserves a piece of memory for the value that the user will enter in the input field. In this example, this memory area has the name `pa_name`. It holds the entry the user makes, but the program can access this value and use it (for example, to output the value using the `WRITE` statement).

Since the contents of this field may change repeatedly, it is sometimes known as a **variable**.

How a Selection Screen Works

When the user enters a value and chooses *Execute*, the input value is transferred to this internal variable. It can then can be used in further calculations or operations. The figure, How a Selection Screen Works, demonstrates how the selection screen and ABAP program interact.

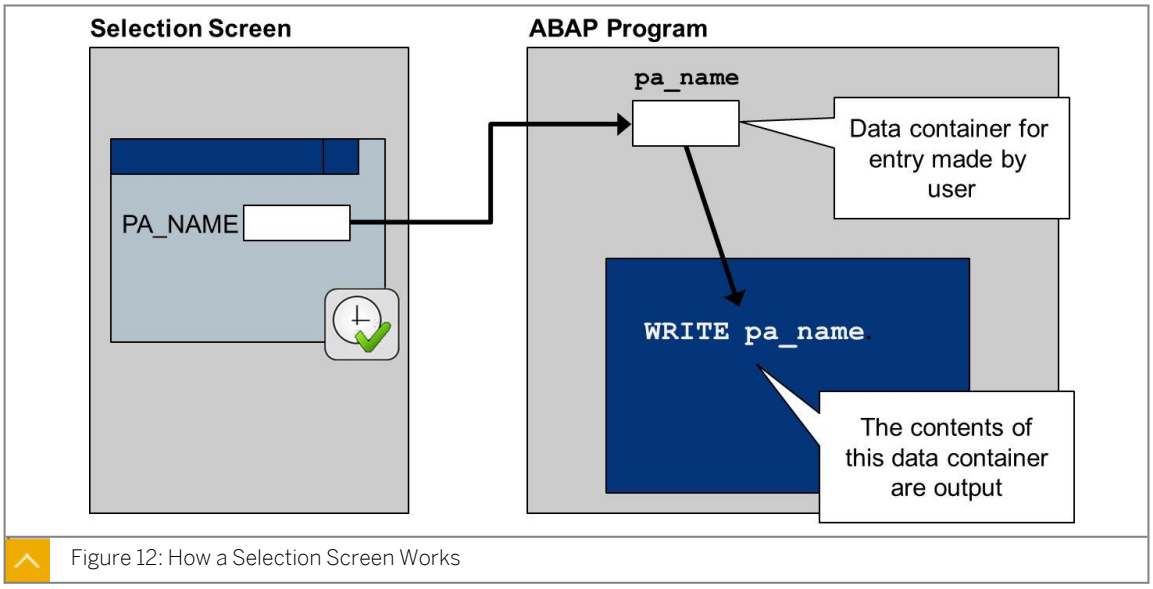

Figure 12: How a Selection Screen Works

You can use selection screens to capture user input for anything that is required by the ABAP program.

Selection screens are often used to enter selection criteria to be used when accessing the database. For example, imagine that you are creating a simple ABAP report that displays sales order data. The users may only be interested in data for certain dates, so you create a selection screen that allows the user to enter the date they are interested in. This value is then considered when the sales order data is retrieved from the database. This has a positive impact on program performance and memory requirement, because only the necessary data will be retrieved.

Multiple Input Fields on a Selection Screen

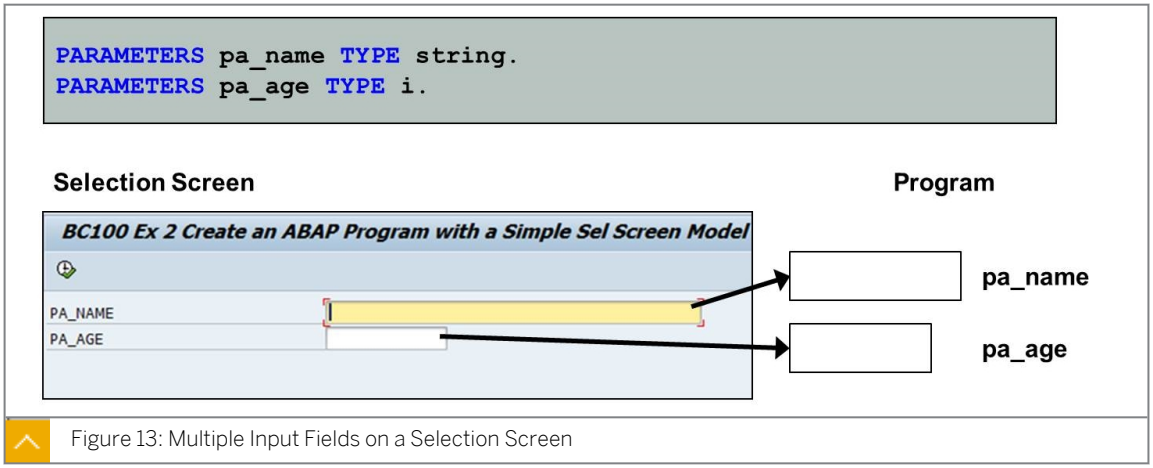

Figure 13: Multiple Input Fields on a Selection Screen

By default, selection screens display the technical names of input fields as their labels (for example, `pa_name` and `pa_age` in the figure Multiple Input Fields on a Selection Screen). You can replace these technical names with more meaningful selection texts, which can also be translated into other languages.

Selection Screen Texts

To maintain selection texts from the ABAP Editor, choose *Goto* → *Text Elements* → *Selection Texts*. To maintain translations for selection texts, choose *Goto* → *Translation*.

The figure, Selection Screen Texts, illustrates the use of selection texts to add translatable labels to input fields.

> Note:
> Selection screen texts are separate objects to the source code. You must activate them separately once created.

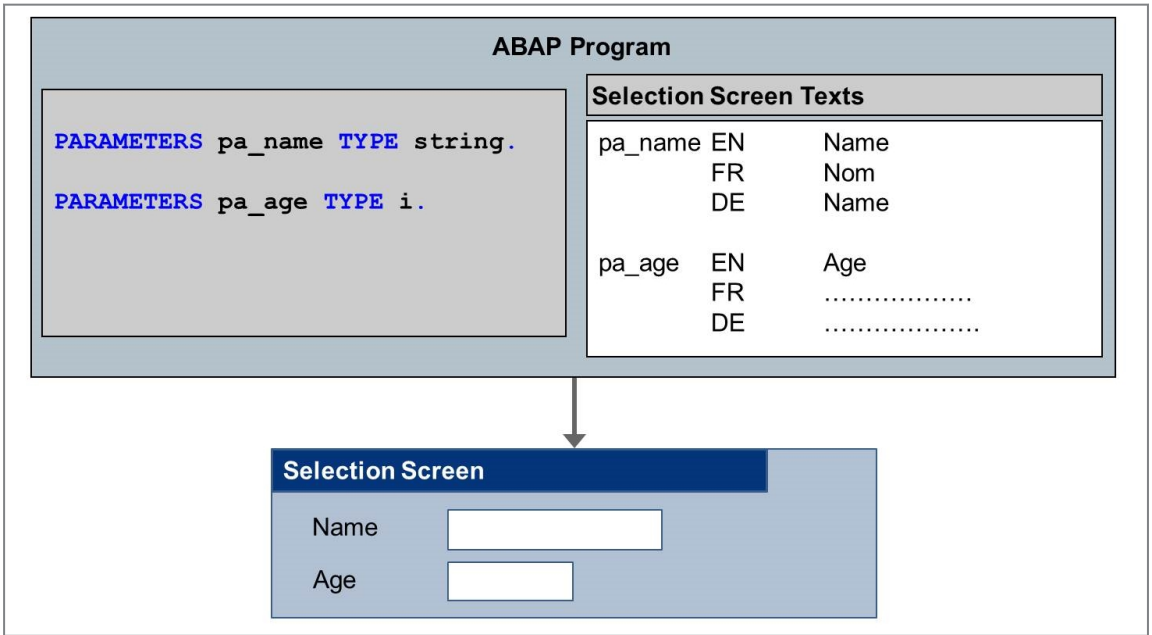

Figure 14: Selection Screen Texts

LESSON SUMMARY

You should now be able to:

- Implement a simple dialog using the PARAMETERS statement

Customizing the ABAP Editor

LESSON OVERVIEW
This lesson shows you how to customize the ABAP Editor to meet your preferences.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Customize the ABAP Editor

ABAP Editor

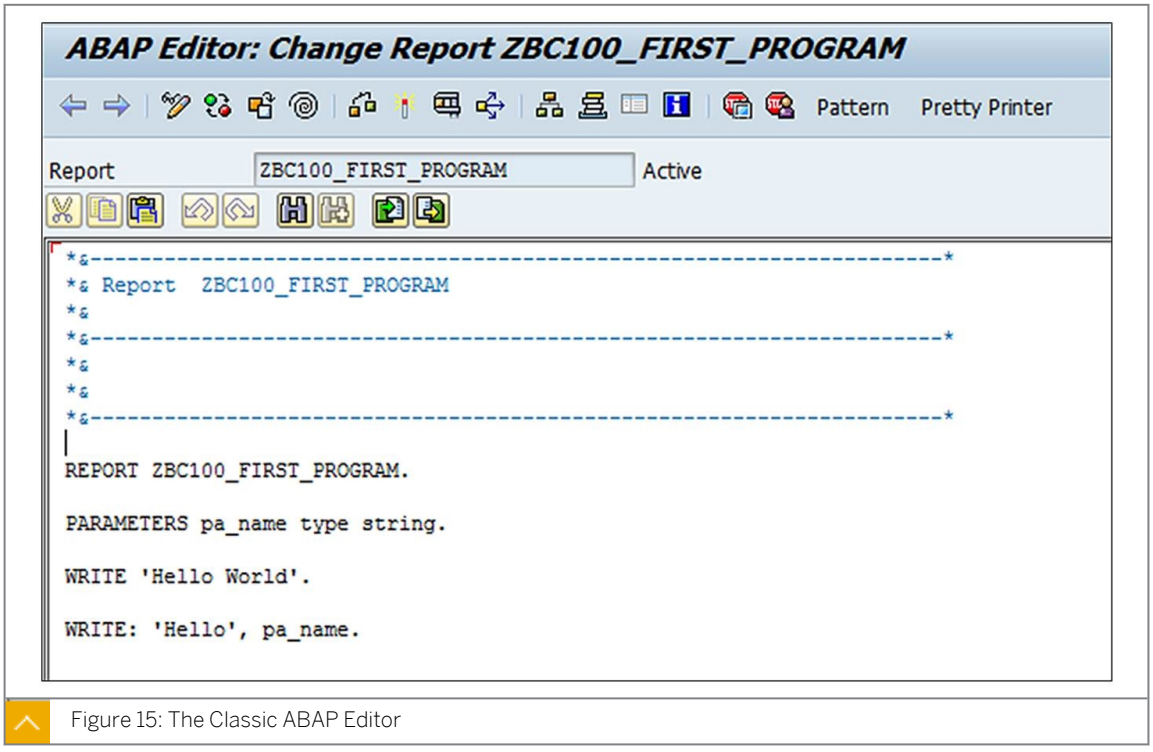

Figure 15: The Classic ABAP Editor

The ABAP Editor has been available for a long time. Some time ago, we also introduced a new, improved ABAP Editor. We recommend that you use the new Editor because it is more flexible and easier to work with. However, you can still use the classic Editor if you prefer.

Choosing Your Preferred ABAP Editor

When you access the ABAP Editor, the classic ABAP Editor might display. If this occurs, you can change to the new ABAP Editor by choosing *Utilities* → *Settings*, then choosing the *ABAP Editor* → *Editor* tab page and selecting the *Front-End Editor (New)* radio button. This option is shown in the figure Choosing the ABAP Editor You Wish to Use.

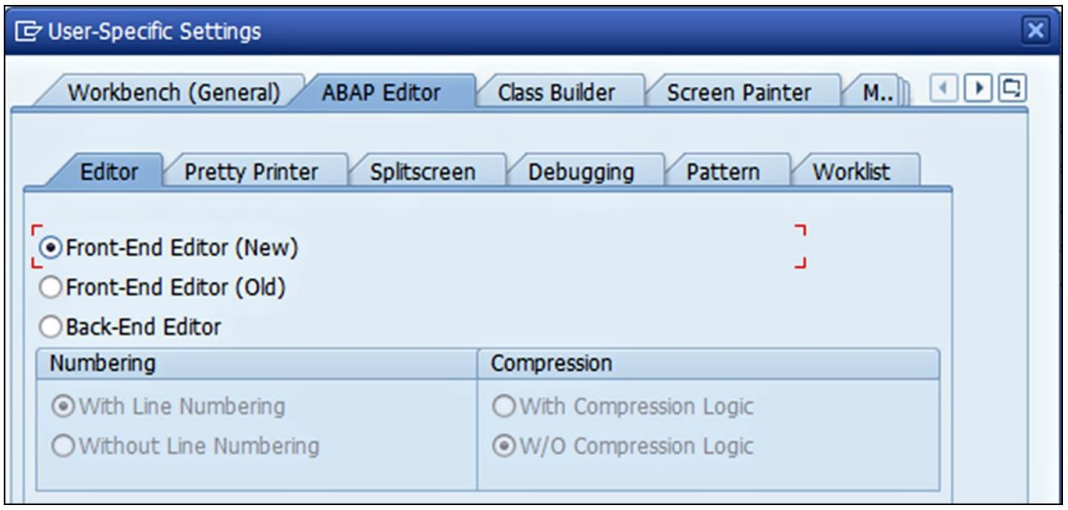

Figure 16: Choosing the ABAP Editor You Wish to Use

Note:
The new Editor is only available if you are working in a system with a suitable release. If not, you can only use the classic ABAP Editor, which is missing some of the new features.

The New ABAP Editor

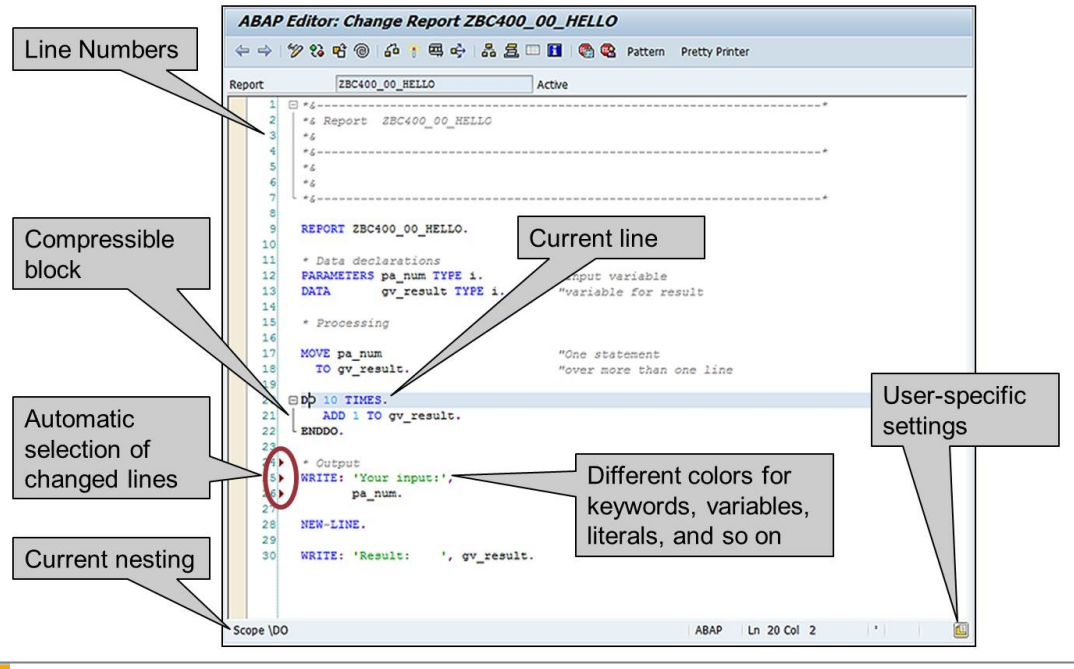

Figure 17: The New ABAP Editor

The new ABAP Editor was developed for SAP NetWeaver 7.0. It can be imported by a support package for SAP NetWeaver Application Server (AS) 6.40 and SAP NetWeaver AS 6.20 (SP 18 or SP 59). The figure The New ABAP Editor illustrates some of the new features, as follows:

- You can specify the display colors for different objects in the source code (such as keywords, comments, and literals).
- You can choose the font and font size to be used for the source code.
- You can use code hints to increase the speed at which you write. The Editor provides code hints for ABAP keywords, and you can also choose to enable code hints for variables.

Pretty Printer

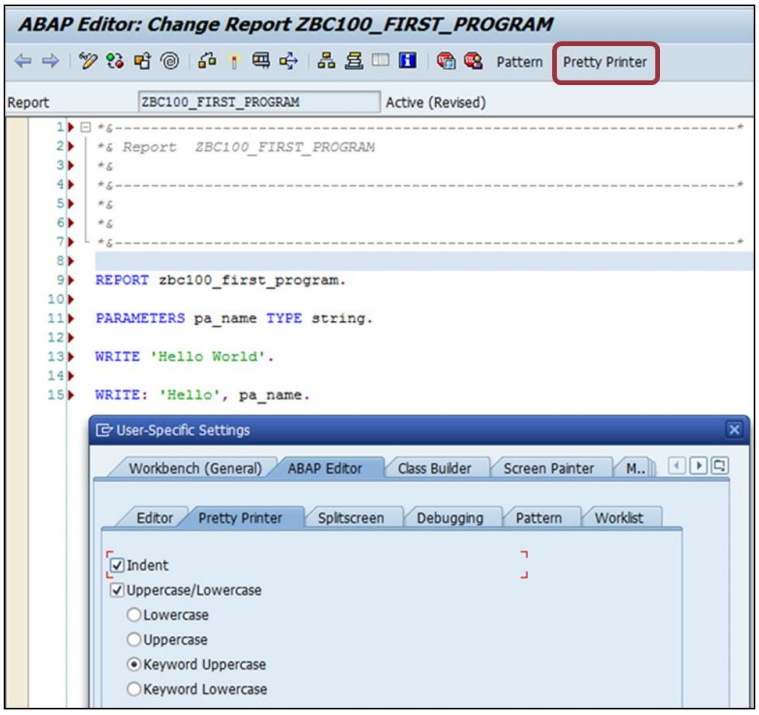

Figure 18: Pretty Printer

Programs that are visually well presented are faster and easier to read, understand, and maintain. All versions of the ABAP Editor include a tool called the Pretty Printer. This can be used (to a limited extent) to format source code and make it easier to comprehend.

The Pretty Printer offers the following possibilities to improve the presentation of your code:

- Keyword capitalization
- Indentation of logically-related blocks

You can change the Pretty Printer settings by choosing *Utilities* → *Settings*, then choosing the *ABAP Editor* → *Pretty Printer* tab page.

The Pretty Printer settings are not applied immediately. You must trigger the process by choosing the *Pretty Printer* button. We recommend that you trigger the Pretty Printer frequently while developing a program.

Note:
If you require more formatting options, such as coloring different code elements or changing font or font size, you must use the new ABAP Editor.

Code Hints and Code Templates

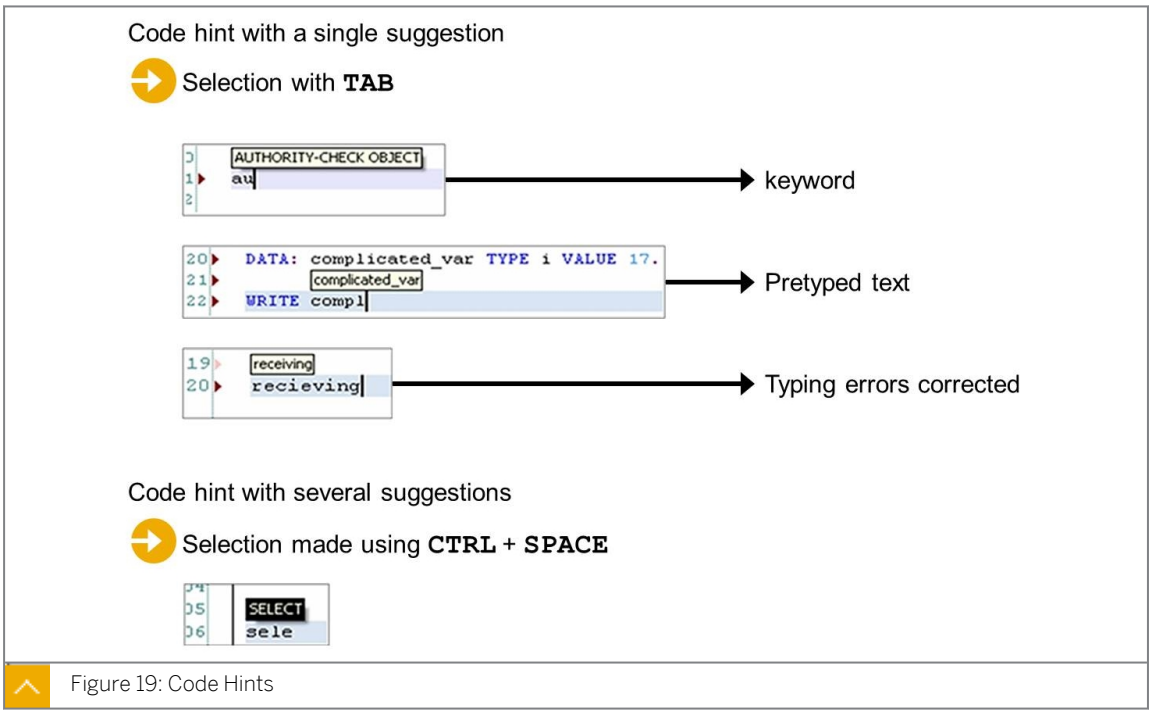

Figure 19: Code Hints

Code hints are one of the main strengths of the new ABAP Editor, because they reduce the amount of typing you need to do. While you type your code, the hints suggest common keywords and recently used identifiers (as shown in the figure, Code Hints).

If one suggestion is available, the code hint displays in black letters on a white background. If there is more than one suggestion available, the most relevant suggestion is shown in white letters on a black background. You can display all available suggestions by pressing **CTRL** + **SPACE**.

Code Templates

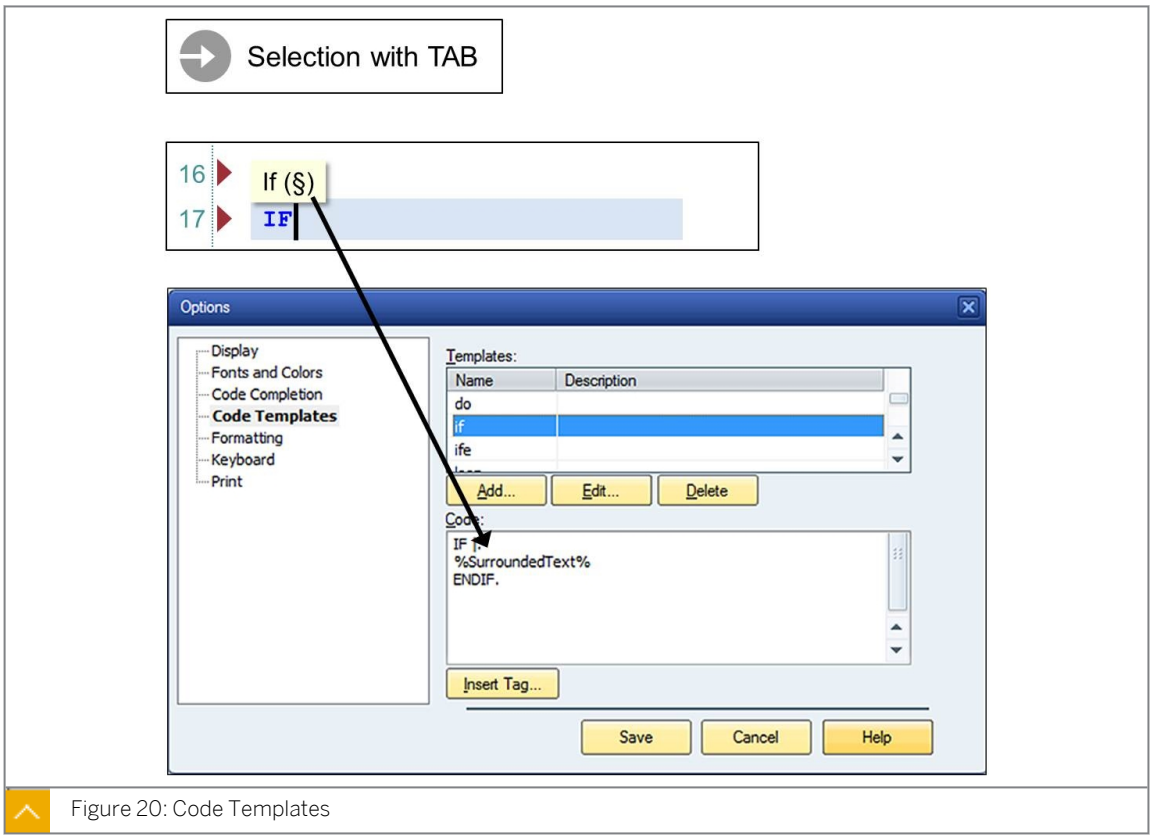

Figure 20: Code Templates

Some code hints contain the symbol § (as shown in the figure Code Templates). These hints denote special Code Templates.

The Editor contains a number of predefined code templates that are suggested by code hints. If you see the § symbol in a code hint, you can insert that predefined template by pressing **TAB**.

The difference between a code hint and a code template is that code hints typically appear for a single keyword, while a code template appears for a block of routine code. For example, the `IF` keyword is never used alone. It is always completed by an `ENDIF` keyword. The Editor contains a code template to help you insert this block of code.

You can also create your own code templates for blocks of code that you regularly use in your own ABAP programs.

Editor Settings

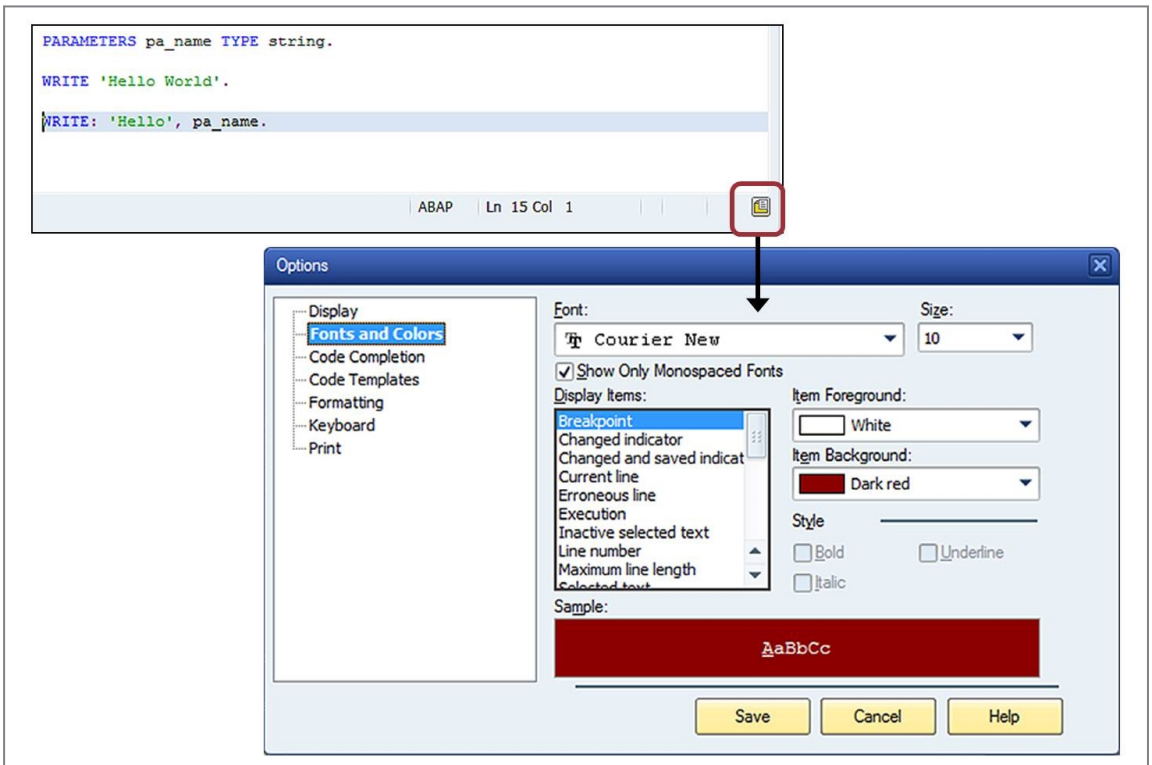

Figure 21: Editor Settings

The Pretty Printer allows you to format your code to a limited extent. The new ABAP Editor also provides more extensive formatting options. You can access these options by choosing the button in the bottom-right corner of the Editor.

The following are some of the display options that you can change:

- The coloring of different code elements
- The font and font size
- The duration for which code hints display on screen

The Editor delivers initial values for all settings, and the settings are user-specific, not system-wide.

Program Generation and Activation

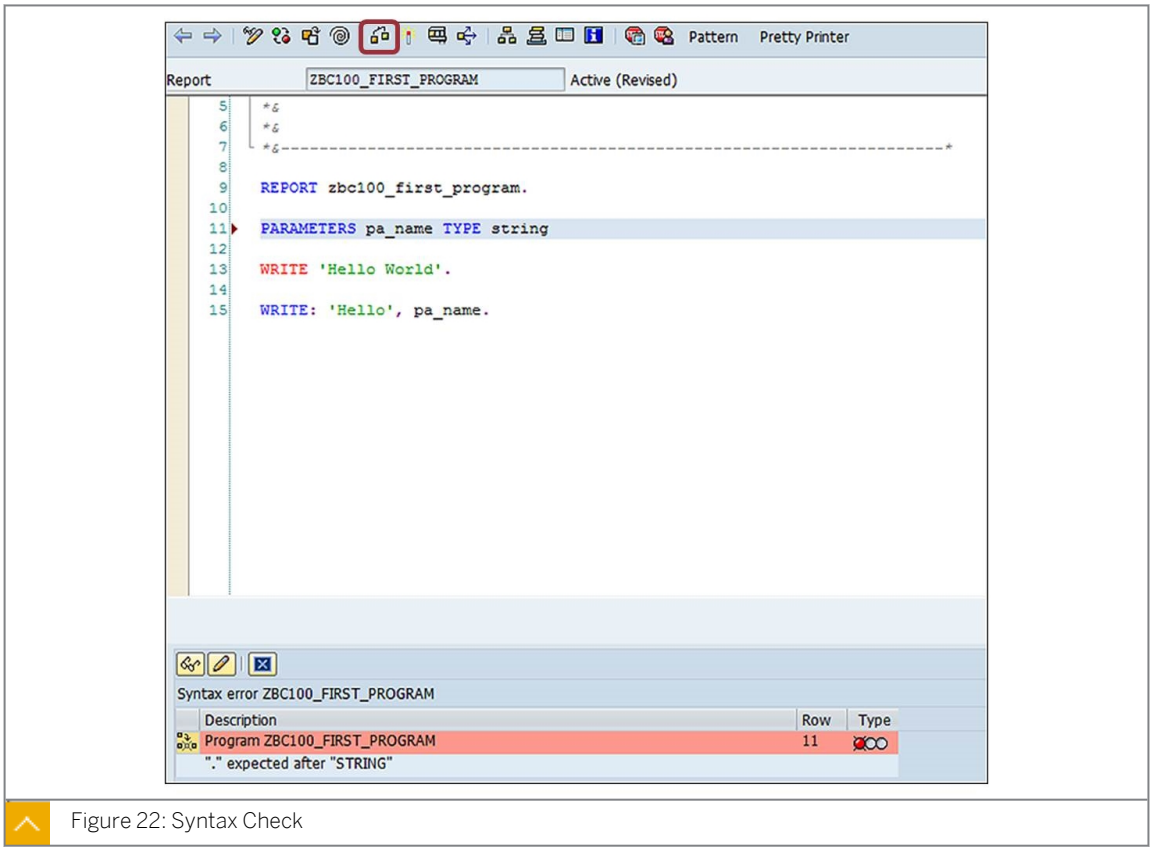

Figure 22: Syntax Check

Regardless of which Editor you use or the way you format your code, remember to save regularly, check your syntax, and activate your programs.

Code which is not well formatted (such as code that does not use indentation or capitalization) can be executed as long as there are no syntax errors. The Syntax Check is like a spellchecker for programs. To check the syntax, choose (*Check*) or press **CTRL** + **F2**. Some of the issues it checks for are the following:

- Do all statements end with a period?
- Are all keywords spelled correctly?
- Does everything between the keyword and the period follow the syntax rules?

Note:
If you use the new Editor, any syntax errors will be highlighted in red in the source code. This indicates that there is an error, but running a syntax check provides more specific information about the error.

Program Activation

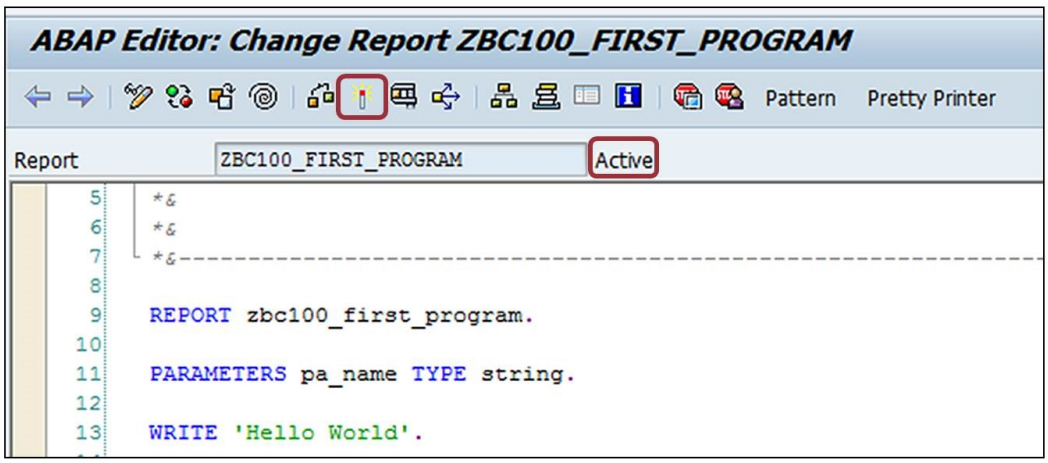

Figure 23: Program Activation

When you have checked the syntax and corrected any errors, activate the program by choosing (*Activate*) or pressing **CTRL** + **F3**.

Every new program starts as inactive. To make it possible for the current version of the program to be accessed by all users, you must activate it.

For example, if a transaction code is assigned to a program, the active version of the program is run when the transaction code is accessed. If you change an active program, a new inactive version is created. Anyone who accesses the transaction code will continue to see the last active version until you activate the new version.

Hint:
If you want to copy a program, ensure that it is active. Only the active version of a program will be copied.

LESSON SUMMARY

You should now be able to:

- Customize the ABAP Editor

Learning Assessment

1. Which naming convention should you use when creating an ABAP program?

   Choose the correct answer.

   - ☐ A There is no naming convention. Use any name you choose.
   - ☐ B The name should start with `Z` or `Y`.
   - ☐ C The name should start with `CUS`.
   - ☐ D The name should start with `PROG`.

2. Which is the punctuation mark that ends an ABAP statement?

   Choose the correct answer.

   - ☐ A Colon
   - ☐ B Comma
   - ☐ C Semicolon
   - ☐ D Period

3. An asterisk (*****) at the start of a program line indicates that the entire line is a comment (and therefore, the ABAP runtime system ignores the entire line).

   Determine whether this statement is true or false.

   - ☐ True
   - ☐ False

4. How can you access the documentation for an unfamiliar keyword in an ABAP program?

   Choose the correct answer.

   - ☐ A Place the cursor on the keyword and press **`F1`**.
   - ☐ B Double-click the keyword.
   - ☐ C Place the cursor on the keyword and choose **`F4`**.
   - ☐ D There is no documentation for ABAP keywords.

5. Which of the following objects are defined by a `PARAMETERS` statement?

   Choose the correct answers.

   - ☐ A Button
   - ☐ B Input field
   - ☐ C Structure
   - ☐ D Variable
   - ☐ E Menu

6. When you finish writing an ABAP program, in which order should you perform these tasks?

   Arrange these steps into the correct sequence.

   - ☐ Activate
   - ☐ Save
   - ☐ Check syntax
   - ☐ Execute program

Learning Assessment - Answers

1. Which naming convention should you use when creating an ABAP program?

   Choose the correct answer.

   - [ ] A There is no naming convention. Use any name you choose.
   - [x] B The name should start with `Z` or `Y`.
   - [ ] C The name should start with `CUS`.
   - [ ] D The name should start with `PROG`.

   You are correct! For custom programs, you must use the customer namespace (that is, use a specific naming convention). Custom program names must begin with Z or Y. For more information, see Unit 1, Lesson 1: Developing a Simple ABAP Program, task Creating Programs in the ABAP Editor.

2. Which is the punctuation mark that ends an ABAP statement?

   Choose the correct answer.

   - [ ] A Colon
   - [ ] B Comma
   - [ ] C Semicolon
   - [x] D Period

   You are correct! Each ABAP statement must end with a period. For more information, see Unit 1, Lesson 1: Developing a Simple ABAP Program, task Basic ABAP Syntax Rules.

3. An asterisk (*****) at the start of a program line indicates that the entire line is a comment (and therefore, the ABAP runtime system ignores the entire line).

   Determine whether this statement is true or false.

   - [x] True
   - [ ] False

   You are correct! The * character at the start of a program line indicates that the entire line is a comment. Therefore, the ABAP runtime system ignores the whole line. For more information, see Unit 1, Lesson 2: Introducing ABAP Syntax, task Keyword Documentation.

4. How can you access the documentation for an unfamiliar keyword in an ABAP program?

   Choose the correct answer.

   [X] A Place the cursor on the keyword and press **F1**.

   [ ] B Double-click the keyword.

   [ ] C Place the cursor on the keyword and choose **F4**.

   [ ] D There is no documentation for ABAP keywords.

   You are correct! To open the documentation for any keyword, place the cursor on the keyword that you want to learn about and press F1. For more information, see Unit 1, Lesson 2: Introducing ABAP Syntax, task ABAP Keyword Documentation.

5. Which of the following objects are defined by a `PARAMETERS` statement?

   Choose the correct answers.

   [ ] A Button

   [X] B Input field

   [ ] C Structure

   [X] D Variable

   [ ] E Menu

   That is correct! The `PARAMETERS` keyword defines an input field on the Selection Screen, and an internal variable with the same name in the memory allocated to the program. When the user enters a value and chooses *Execute*, the input value is transferred to this internal variable. For more information, see Unit 1, Lesson 3: Implementing a Simple Dialog, task The Parameters Keyword.

6. When you finish writing an ABAP program, in which order should you perform these tasks?

   Arrange these steps into the correct sequence.

   [3] Activate

   [1] Save

   [2] Check syntax

   [4] Execute program

   You are correct! Regardless of which Editor you use or the way you format your code, save, check syntax, activate and execute your programs following this sequence. Syntax check can be executed after you save the source code. The program can be activated only if there are no syntax errors. Then, the program can be executed. For more information, see Unit 1, Lesson 4: Customizing the ABAP Editor, task Program Generation and Activation.

UNIT 2 Coding and Debugging in ABAP

UNIT OBJECTIVES

- Describe data types
- Declare variables and constants
- Create text symbols
- Perform basic arithmetic operations in ABAP
- Use system variables in a program
- Debug a program
- Create an ABAP list

- Process character strings in a program

Defining Simple Variables

LESSON OVERVIEW
In this lesson, you learn about the use of data types, variables, and constants in ABAP.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Describe data types
- Declare variables and constants

Use of Data Types

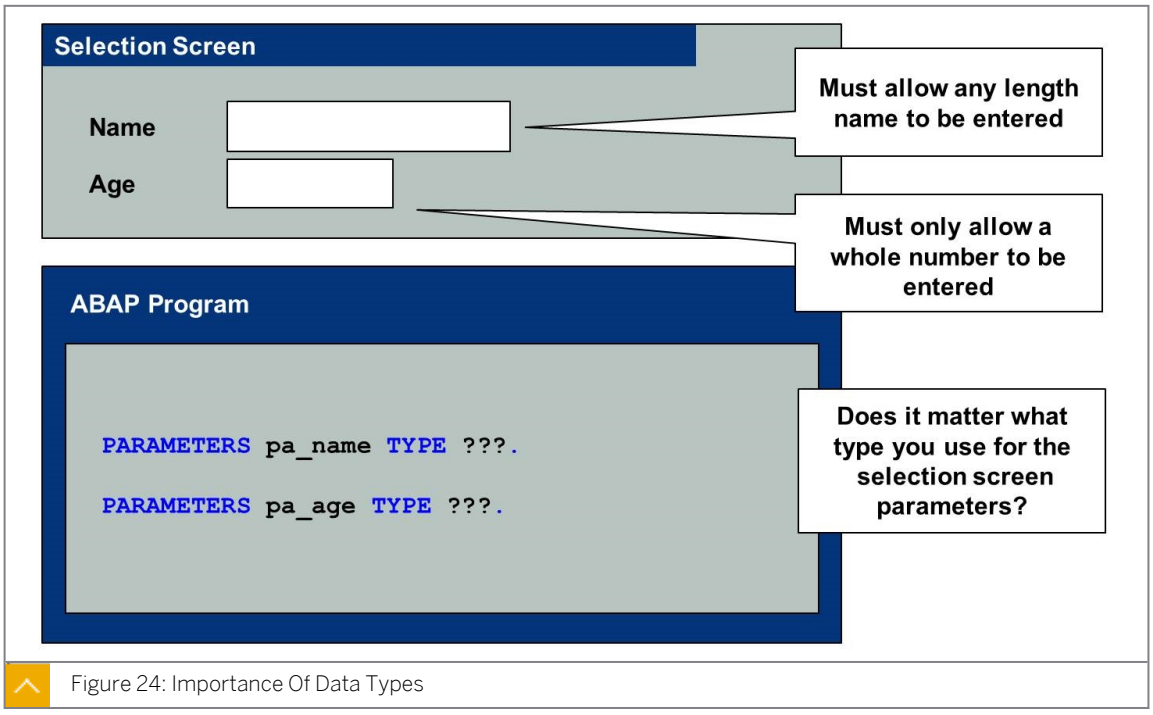

Figure 24: Importance Of Data Types

Why does the type of a selection screen parameter matter?

The `PARAMETERS` statement declares an input field that is also a variable in the program. When it is defined, the parameter must be given a name and also a specific type.

Choosing Data Types for Selection Screen Parameters

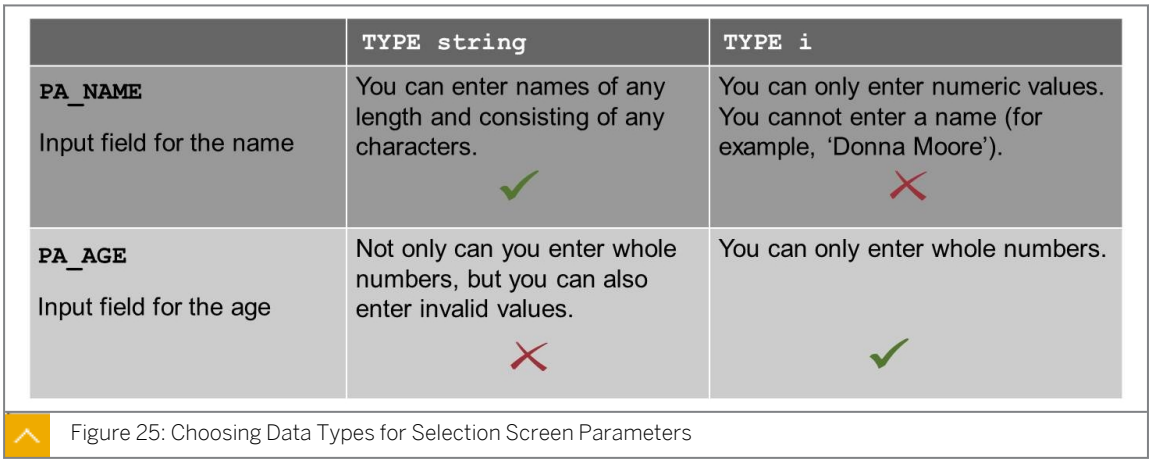

Figure 25: Choosing Data Types for Selection Screen Parameters

The name of the parameter can be chosen freely. It has no impact on the values that can be entered in the field. The type of the parameter determines the values that can be entered (numbers, characters, strings, dates, and so on) and how large those values can be. You can think of a data type as being a formal description of the variable (input field). The figure Choosing Data Types for Selection Screen Parameters shows how different data types are appropriate for different parameters, depending on the intended use.

The data type does not just determine the values that can be entered in the input field. It also impacts what can be done with the data in the ABAP program. For example, if a numeric data type has been used (such as type `i` in the table), then calculations can be performed using that data. If a variable is type `string` and contains the value 'ABC', calculations cannot be performed.

It is important to consider which data type to give to a selection screen parameter, as this will determine which values the user can enter in the corresponding input field. It is also important to consider the correct data type to give to internal program variables. The declaration and use of these variables is covered in another lesson.

ABAP Standard Data Types

Table 2: Character-like and Numeric Data Types

Type	Meaning	Use
	Character-like Types	
string	Variable length character string	Any characters. Has an arbitrary length.
c	Fixed-length character string	Any characters. Has a fixed length.
d	Date	Holds dates in the format YYYYMMDD.
t	Time	Holds times in the format HHMMSS.
	Numeric Types	
i	Integer	Whole numbers (no decimal places).
p	Packed number	Decimal numbers.

Character-like Types

Variables of type `c` or type `string` can both hold character strings. Variables of type `string` can be any length. You do not need to specify a length and it is not possible to do so. Variables of type `c` have a specific length; you specify this using the `LENGTH` addition as follows:

```
DATA gv_var1 TYPE c LENGTH 2.

* Or

PARAMETERS pa_field TYPE c LENGTH 10.
```

Type `d` is used to hold dates. In ABAP, a date is always stored internally in the format YYYYMMDD. If a user enters a value in an input field that has the type `d`, the system checks the input format and validity of the date. However, the user does not need to enter the date in the internal format. Every user profile is assigned a default date format (for example, DD.MM.YYYY). The user can enter a date in the input field in this format, and the system converts the date to the internal format. Since a date variable always contains eight digits, you cannot specify the length.

Variables of type `t` are used to hold times. These are stored internally in the format HHMMTT. Since a time variable always contains six digits, you cannot specify the length.

Numeric Types

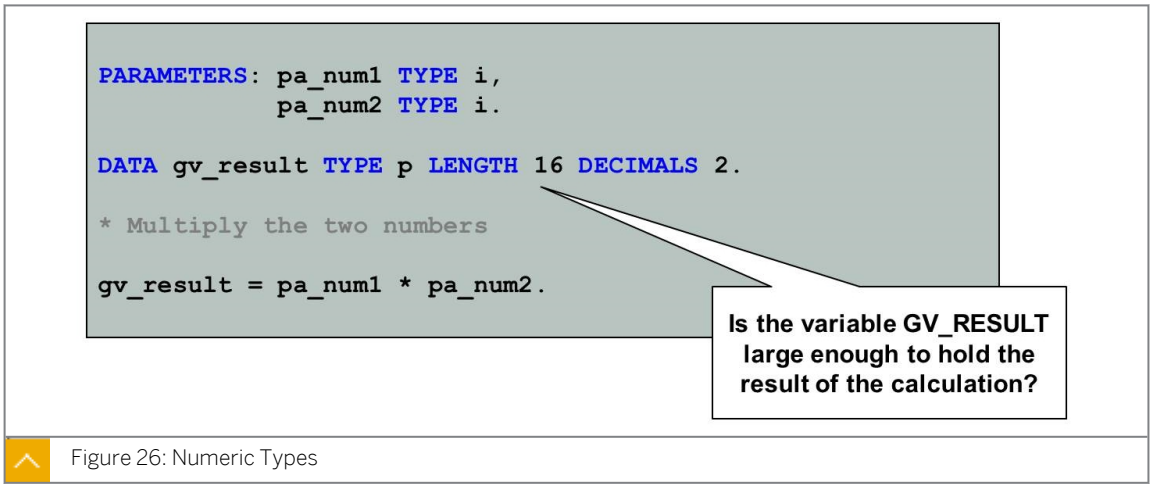

Figure 26: Numeric Types

Type `p` is one of the most important and common ABAP data types. It can be used for business calculations where the result must be accurate.

The abbreviation `p` comes from the term ‘packed’, because this data type packs two digits into each byte. In principle, a variable of type `p` can contain twice as many digits as the specified length, but half a byte is required for the sign (since negative values can be stored in a variable of type `p`).

For example, a packed number with a length of 16 has 32 available spaces. One space is reserved for the sign, which leaves 31 for the remaining digits.

By default, a variable of type `p` has no decimal places. If decimal places are required, declare them using the `DECIMALS` addition.

Note:
A variable of type `p` does not contain a thousand separator. Thousand separators (for example, 1,000) are not part of the variable value. However, they display automatically if the variable is output to the screen.

Unlike type `p`, the length of a variable of type `i` is fixed. Since only integers are allowed, you cannot specify a number of decimal places in the declaration. This means that for type `i` variables, you do not need to supply anything more after the type specification.

Global Data Types

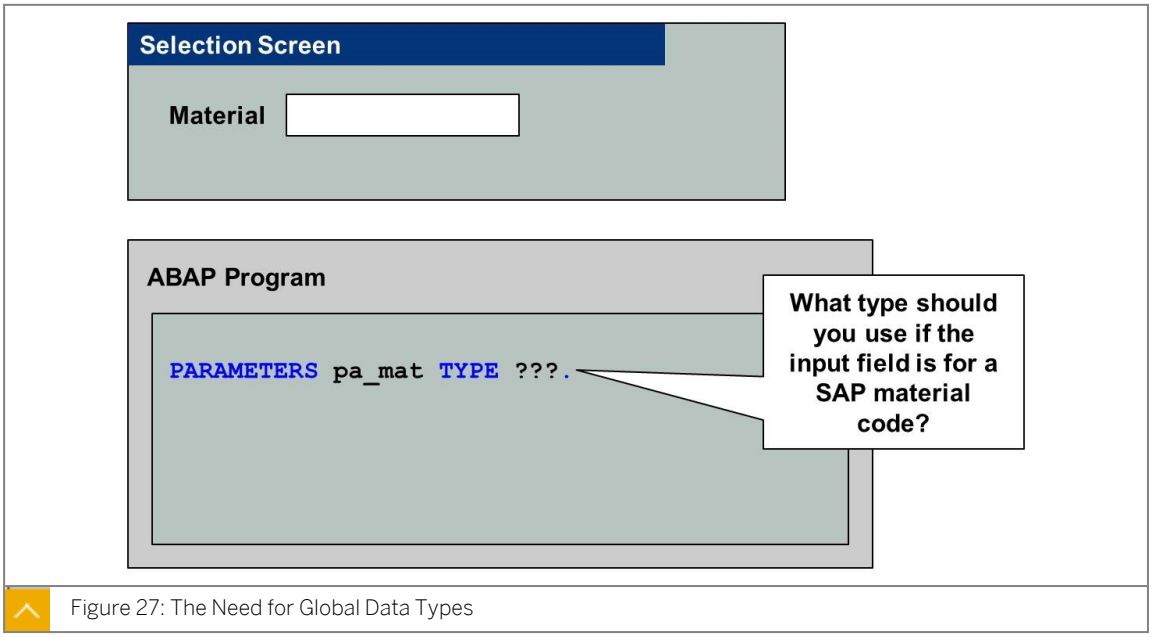

Figure 27: The Need for Global Data Types

The figure The Need for Global Data Types illustrates the importance of data types. How long is a material code? Can it contain letters, or numbers only?

The ABAP Dictionary

Many business entities that you may need to work with in your ABAP programs are described in the SAP system. These descriptions can be found in the ABAP Dictionary by looking at the definitions of the database tables that hold your SAP data.

For example, material master data is stored in the table MARA. By opening the description of the table in the ABAP Dictionary (transaction code `SE11`) you can see that the column that represents the material number is called MATNR. You can also see that the field is a character string of 18 characters in length.

If you need a variable or input field to hold a material code in your ABAP program, the easiest way to do this is to use the type `table-columnname` as follows:

```
PARAMETERS pa_mat TYPE MARA-MATNR.
```

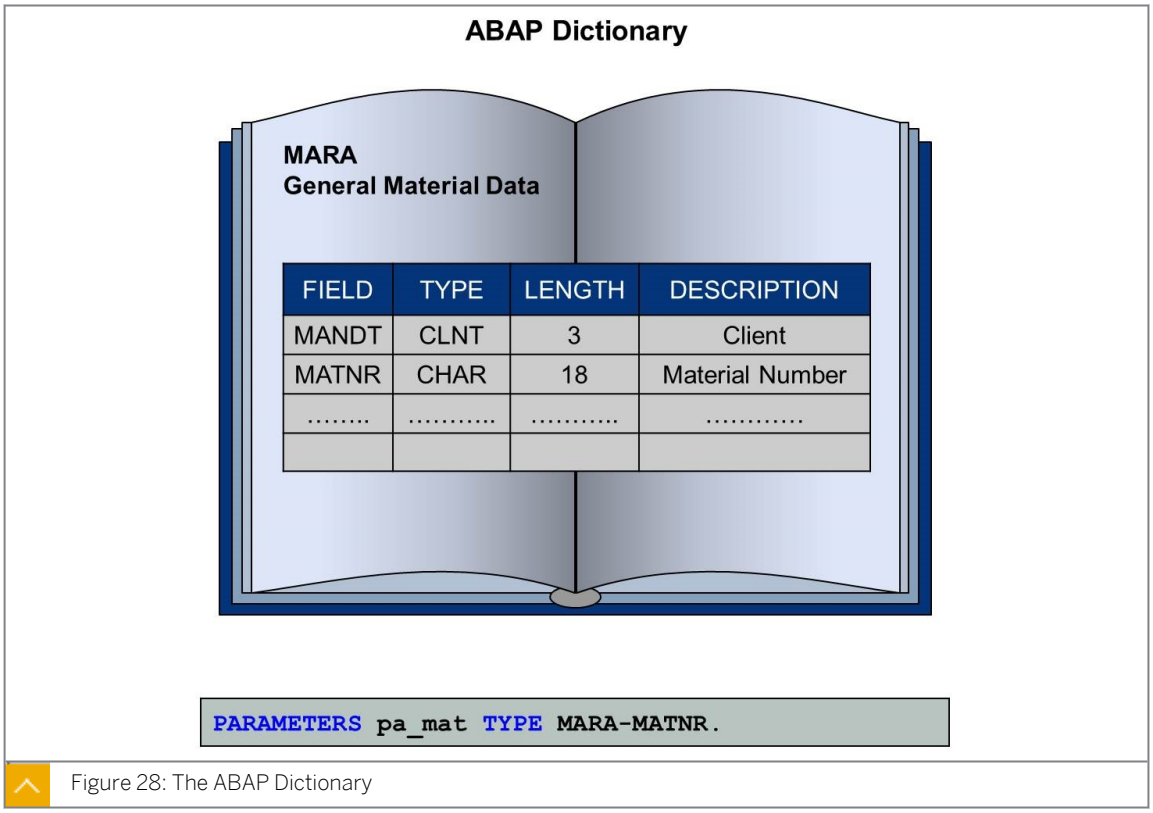

Figure 28: The ABAP Dictionary

The figure The ABAP Dictionary illustrates the use of table fields to specify the type of an input field. ABAP Dictionary table fields are globally visible. They are known and can be used throughout the entire system, which means that you can use them in any ABAP program.

Note:
This lesson focuses only on the use of table fields and how to use them as the types for variables or input fields. Other global data types are available in the ABAP Dictionary and are explained in the course ABAP Workbench Foundations (BC400).

Variables

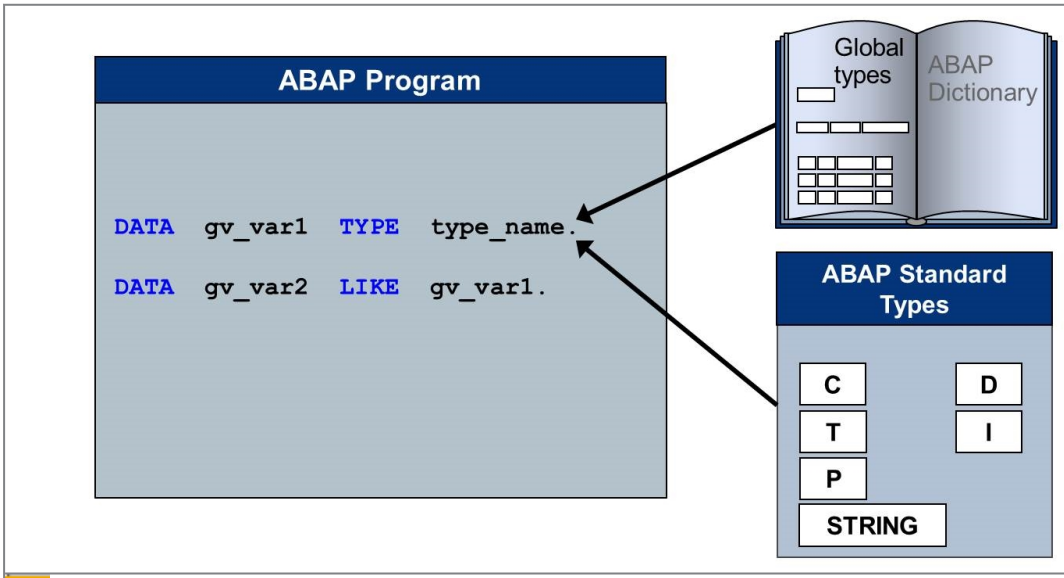

Figure 29: Defining Variables

You can define a variable using the `DATA` keyword. To provide a description or type for the variable, you can use either an ABAP standard type or a global type from the ABAP Dictionary. You can also use the `LIKE` addition to refer to an existing variable when defining additional variables.

Do not confuse the `DATA` keyword with the `PARAMETERS` keyword. `PARAMETERS` defines an input field on a selection screen that is also a variable whose value can be accessed in the program. `DATA` defines a variable that is accessible and will be used internally within the program. It is not associated with a user input field.

Variable Declarations

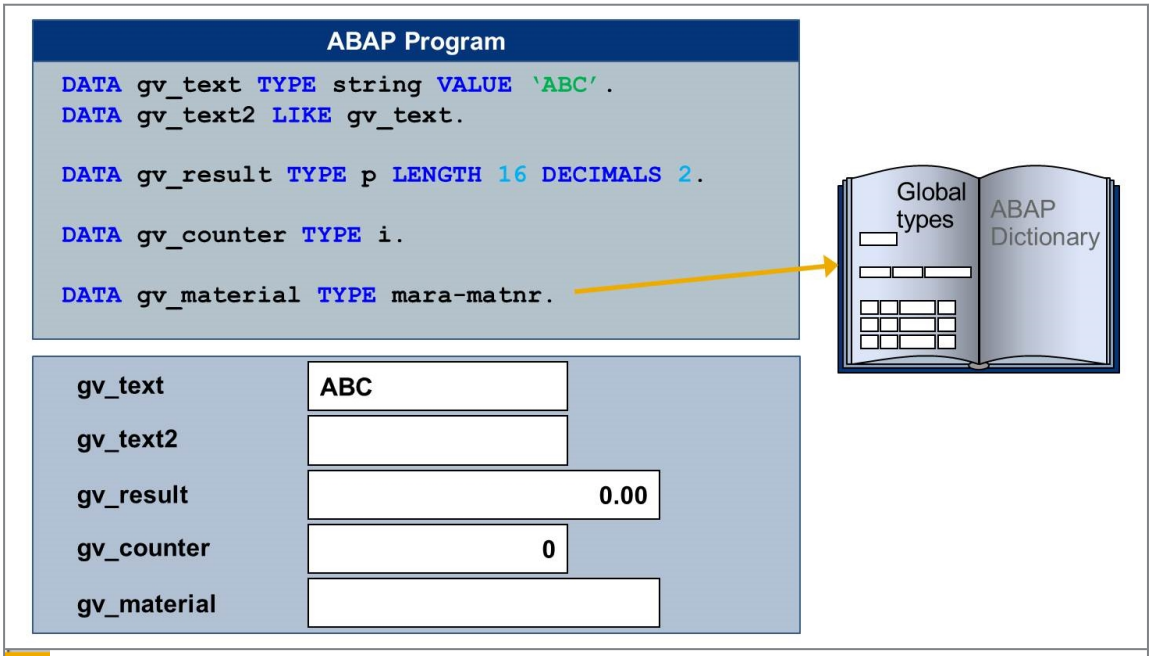

Figure 30: Example Variable Declarations

You can use the VALUE addition to preassign a value to a variable. In the DATA statement, there are two ways that you can provide the length for any variable of a type that requires it (for example, type c and type p). The two ways are as follows:

```
DATA gv_result TYPE p LENGTH 16 DECIMALS 2.

* Or

DATA gv_result(16) TYPE p DECIMALS 2.
```

If you do not specify the length in a variable definition, a default length for the standard type is used. The default length is 1 for type c and 8 for type p.

Literals and Constants

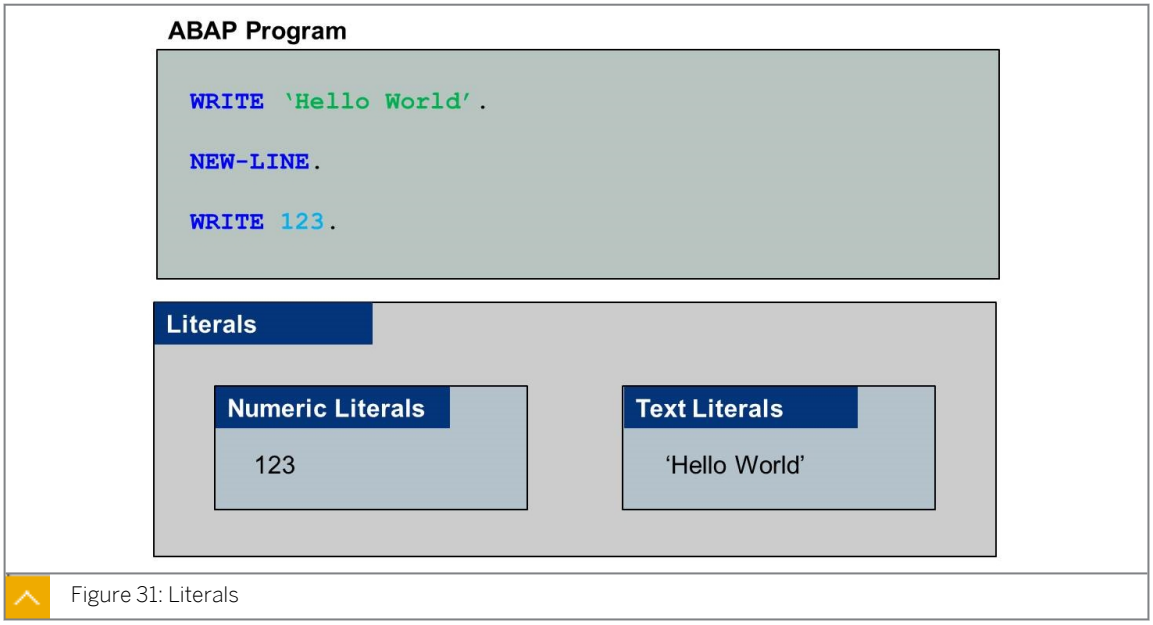

Figure 31: Literals

Literals

In the figure Literals, Hello World and 123 are known as **literals**.

Literals are strings of characters without a name. Their values cannot be changed, since they are essentially hard-coded values. Numeric literals consist of continuous sequences of numbers, and text literals are character strings.

Numeric literals can be specified without quotes, whereas text literals must be enclosed in single-quotes, as shown in the figure.

Since text literals cannot be changed, you cannot translate them into other languages. This can cause problems. A solution is to use text symbols instead of text literals. The lesson Defining Text Symbols provides more information about text symbols.

Constants

```
CONSTANTS c_hello TYPE string VALUE 'Hello World'.
CONSTANTS c_number TYPE i VALUE 123.

WRITE c_hello.

NEW-LINE.

WRITE c_hello.
```

Figure 32: Constants

We recommend that you avoid using literals to specify values in your source code. Instead, define constants with those values and use the constants in place of literals.

One reason for this is that certain values will be required in more than one place in the source code. It is inconvenient to specify these values directly, since you would need to change several statements if the value needs to be changed. Instead, you can use an appropriate constant so that you only need to modify the value once. This can make a program easier to maintain and enhance in the future.

It can also be a good idea to create an appropriate constant for values that are used in one place only. Giving the constant a meaningful name makes it easier to understand the source code.

The value of a constant cannot be changed at runtime. Instead, its value is assigned in the definition using the `VALUE` addition. The figure Constants demonstrates the declaration of two constants (one text and one numeric) and the use of the constants in a statement.

LESSON SUMMARY

You should now be able to:

- Describe data types
- Declare variables and constants

Defining Text Symbols

LESSON OVERVIEW
This lesson shows you how to create a text symbol and use the multilingual capability of ABAP.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Create text symbols

Text Symbols

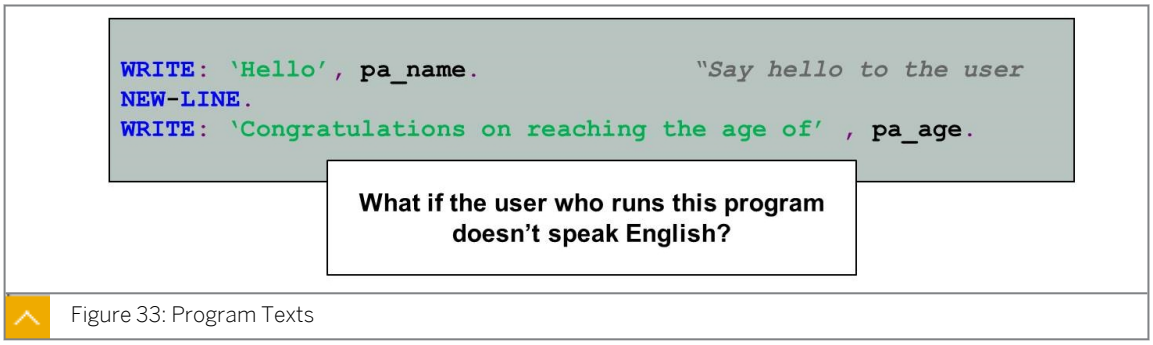

Figure 33: Program Texts

Multilingual capability is an important principle in ABAP development. It means that the logon language of the current user is accounted for when texts display on the user interface. For example, a user logged on in English will see texts that are written in English, while a user logged on in French will see the same texts written in French.

As a consequence, we recommend that you avoid the use of text literals (hard-coded texts) when developing programs, since a literal exists in only one language in the source code. If you are developing productive programs that will be executed by different users in different logon languages, use text symbols to ensure that all texts are translatable.

Text Symbols

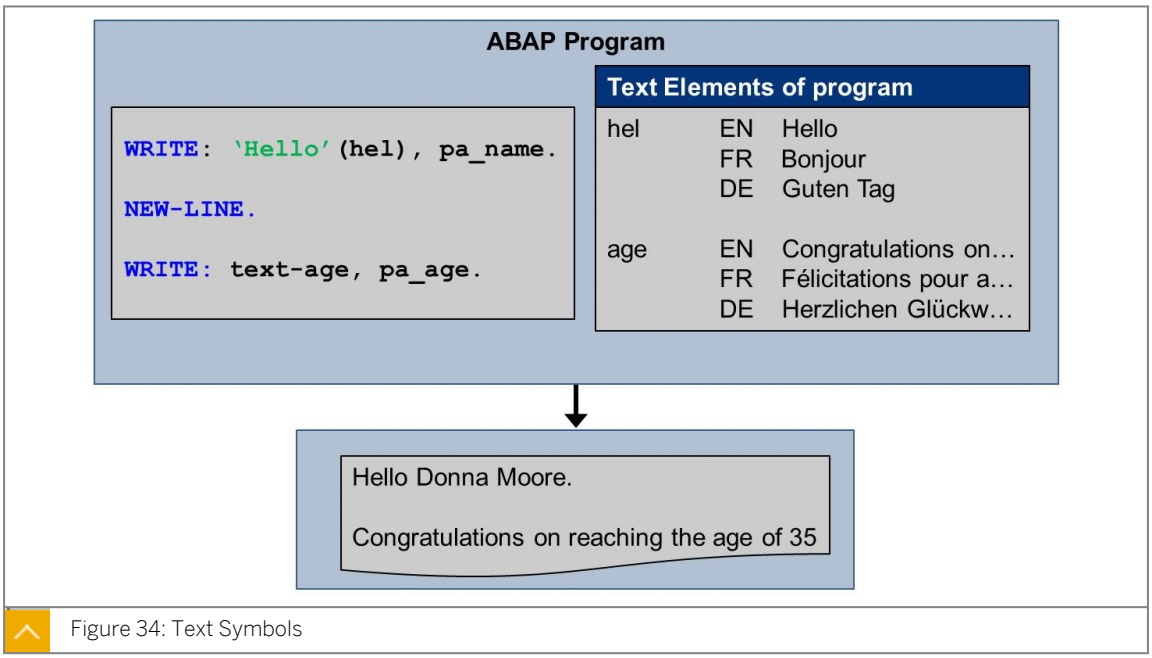

Figure 34: Text Symbols

Text symbols belong to a specific program and can be translated into different languages. When a statement using a text symbol is executed, the system automatically takes into account the logon language of the user and supplies the text in this language. This feature is illustrated in the figure Text Symbols.

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

There are two ways to address text symbols in your code:

- `TEXT-xxx`, where `xxx` stands for the three-character text symbol ID.

  For example, `WRITE TEXT-abc.`

- `'...'(xxx)`, where the ellipsis (...) is the original language text.

  For example, `WRITE 'Hello'(abc).`

There are two ways to access the screen where you can define a text symbol:

- Choose *Goto* → *Text Elements* → *Text Symbols*.
- Define the text symbol ID in the source code, then double-click it.

  The Editor asks you if you want to create a text symbol, then opens the *Text Symbols* screen

You can maintain translations for text symbols on the *Text Elements* screen. Choose *Goto* → *Translation* and enter the relevant language and translation for the text symbols.

Note:
Remember to activate text elements. They are a separate program object from the source code.

LESSON SUMMARY

You should now be able to:

- Create text symbols

Unit 2
Lesson 3

Performing Arithmetic Operations Using Simple Variables

LESSON OVERVIEW
In this lesson, you learn how to perform arithmetic operations in an ABAP program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Perform basic arithmetic operations in ABAP

Value Assignment to Simple Variables

```
DATA: gv_num TYPE i,
      gv_num2 TYPE i,
      gv_text TYPE string,
      gv_decimal TYPE p LENGTH 16 DECIMALS 2.
```

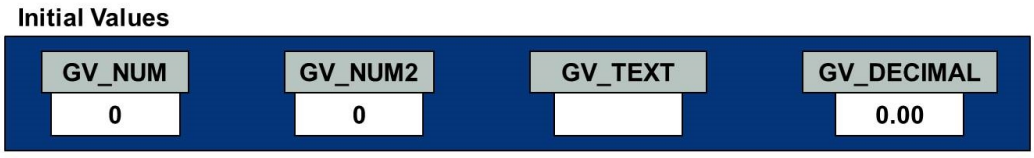

Figure 35: Initial Values Of Variables

You now understand how to define variables to hold the data that your program works with. Next, we will examine how to work with them.

When a program starts, the system makes memory available for the variables defined in the program. Every variable is preassigned an initial value based on its type unless you set a different value using the `VALUE` addition. For example, the initial value of a numeric variable is 0, and the initial value of a character-type variable is a sequence of spaces. The figure Initial Values of Variables demonstrates the initial values of different variable types.

Value Assignments

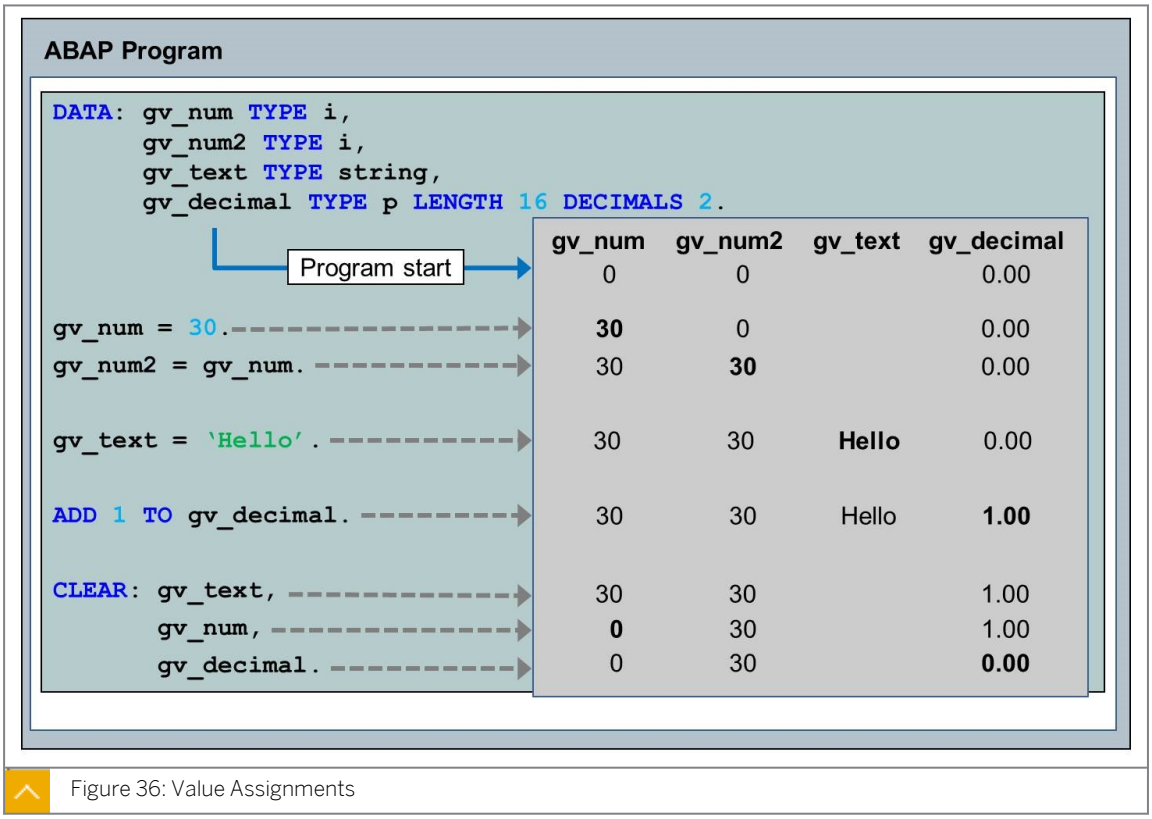

Figure 36: Value Assignments

You can use several different ABAP statements to fill, assign values to, or change the value of variables. The figure Value Assignments demonstrates how the values of a set of variables change as the program processes several statements.

For example, you can use the following syntax to assign a value to a variable:

```
gv_num = 30.
```

The target variable is to the left of the equal sign. The value to the right of the equal sign will be assigned to the target variable and will overwrite the current value.

You can also use the `MOVE` statement to transfer the contents of one variable to another, as follows:

```
MOVE gv_num TO gv_num2.
```

This has the same result as using the short-form assignment. In this example, the `MOVE` statement is equivalent to the following assignment:

```
gv_num2 = gv_num.
```

The `CLEAR` statement resets the contents of a variable to the type-related initial value. For more information about the initial values for a type, see the keyword documentation for the `CLEAR` statement.

Calculations Using Simple Variables

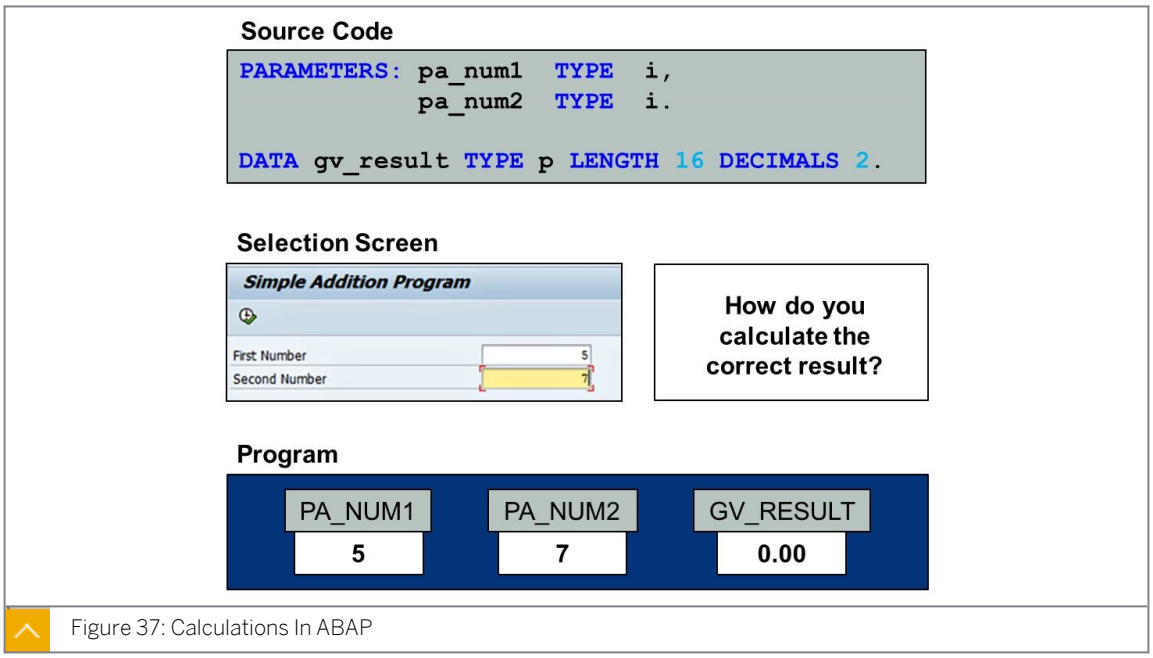

Figure 37: Calculations In ABAP

The figure Calculations in ABAP shows an example of a program where the user enters two numbers, which are then used to perform a calculation.

Result of Calculations in ABAP

We have seen some ABAP keywords that you can use to assign values to variables, but direct assignment is not always appropriate. Sometimes you want to fill your variables with the result of a calculation.

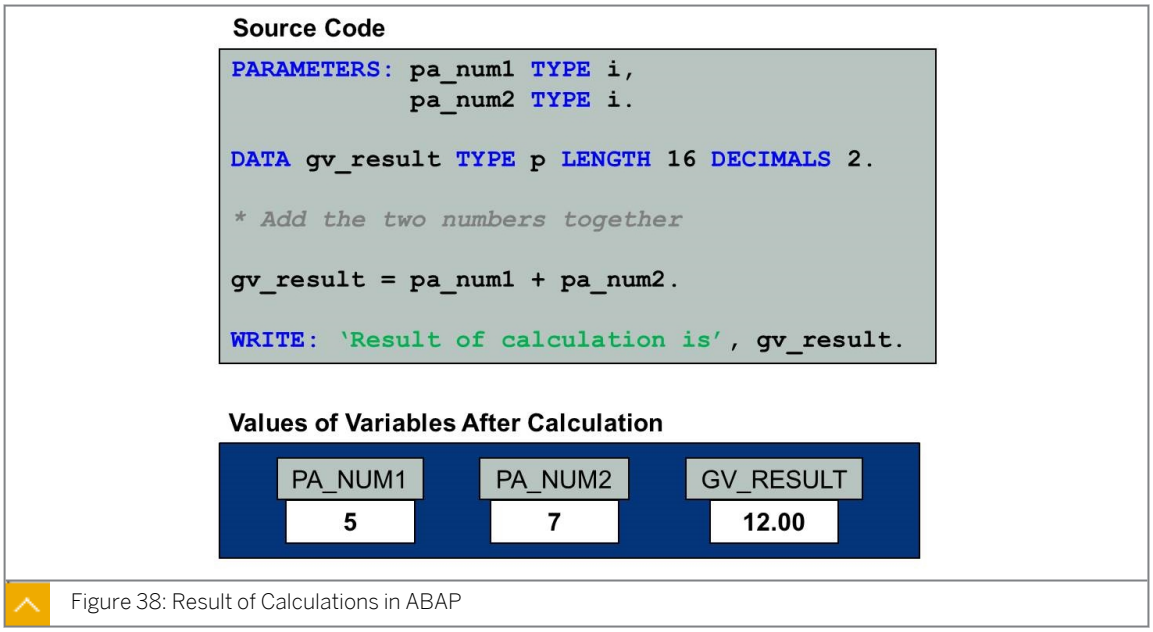

Figure 38: Result of Calculations in ABAP

Arithmetic Operators for ABAP Calculations

ABAP allows you to program arithmetic operations easily. The list of valid operators for an arithmetic operation includes the following:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)

You can use parentheses in the calculation if one arithmetic expression must be used as the operand for another arithmetic expression, as follows:

```
gv_result = ( pa_num1 / pa_num2 ) * 100.
```

Note:
For more information about the way in which calculations are performed, see the ABAP keyword documentation on Arithmetic Expressions.

Table 3: Arithmetic Operators for ABAP Calculations

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division

Keyword Alternatives for Arithmetic Operators

Table 4: Keyword Alternatives for Arithmetic Operators

Keyword	Example
ADD	ADD 1 TO gv_result.
SUBTRACT	SUBTRACT 1 FROM gv_result.
MULTIPLY	MULTIPLY gv_result BY 2.
DIVIDE	DIVIDE gv_result BY 4.

As shown in the table Keyword Alternatives for Arithmetic Operators, you can also use specific keywords as an alternative to using the operators in a calculation.

For example, to increment a variable, you could use either of the following alternatives:

```
gv_result = gv_result + 1.

* Or

ADD 1 TO gv_result.
```

LESSON SUMMARY

You should now be able to:

- Perform basic arithmetic operations in ABAP

Unit 2
Lesson 4

Using System Variables

LESSON OVERVIEW
This lesson shows you how to use system variables in your program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Use system variables in a program

System Variables

Table 5: Frequently Used System Variables

System Variable	Description
`SY-DATUM`	Current date
`SY-UZEIT`	Current time
`SY-UNAME`	User ID of current user

So far, you have declared all variables that you work with in your ABAP programs. However, the system contains special **system variables** (or system fields) that you can use in your source code without explicitly declaring them. The ABAP runtime system manages these system variables, and they are available to any ABAP program that wants to use them.

Note:
You can recognize a system variable by its name. All system variable names begin with `SY-`.

System variables provide information about the actual system status. The ABAP runtime system populates and changes the values of the system fields when necessary. For example, the system variable `SY-DATUM` is always filled with the current date, and `SY-UZEIT` is filled with the current time.

The table System Variables shows just three system variables. As you learn more about ABAP, you will learn about other important system variables. For a complete list of system variables, see the ABAP keyword documentation for System Fields.

Note:
System variables are intended for read access only.

LESSON SUMMARY

You should now be able to:

- Use system variables in a program

Debugging a Program

LESSON OVERVIEW
In this lesson, you learn how to identify and correct bugs using the ABAP Debugger.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Debug a program

ABAP Debugger

The ABAP Debugger is an important diagnostic tool that you can use to analyze ABAP programs.

With the Debugger, you can determine why a program is not working correctly by 'stepping inside' the program at runtime. This allows you to see the statements being executed and the changing values of variables as the program proceeds.

To use the Debugger, you first decide at which point you want to activate it. Depending on your needs, there are different ways to start the Debugger.

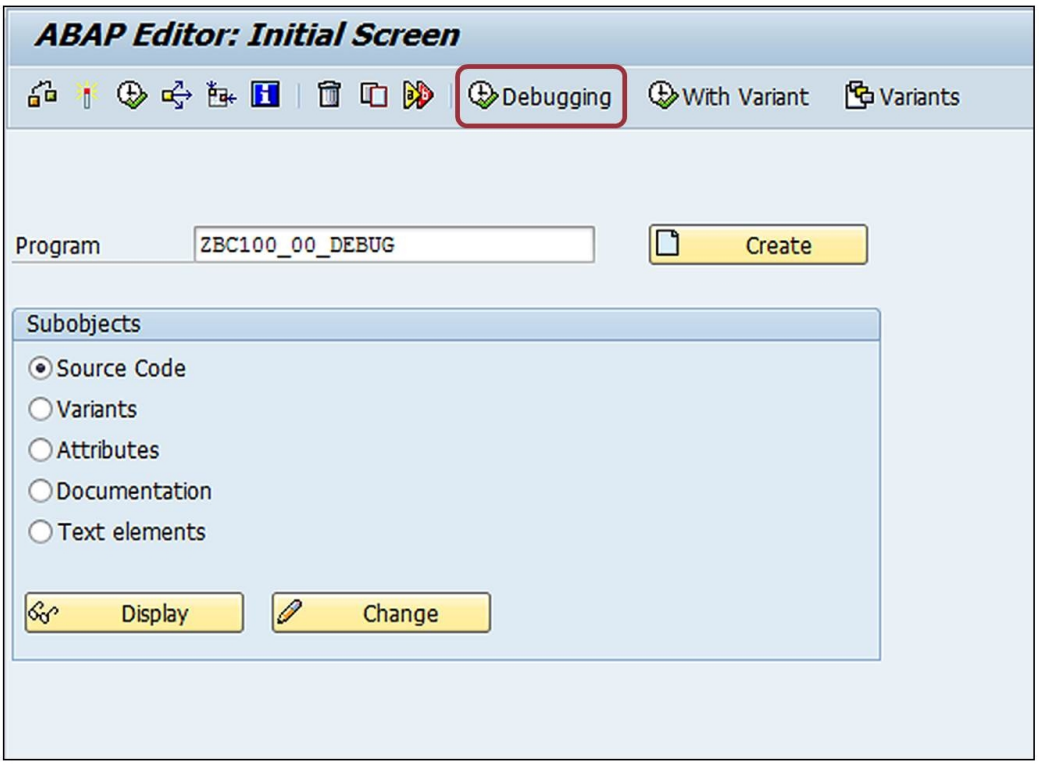

Figure 39: Debugging a Program From the Beginning

If you want to get a general understanding of how a program works, you can start the Debugger from the very beginning (that is, from the initial screen of the ABAP Editor). Starting the Debugger here allows you to follow the program logic from start to finish. To start the Debugger from this screen, choose the *Debugging* button.

Setting a Breakpoint

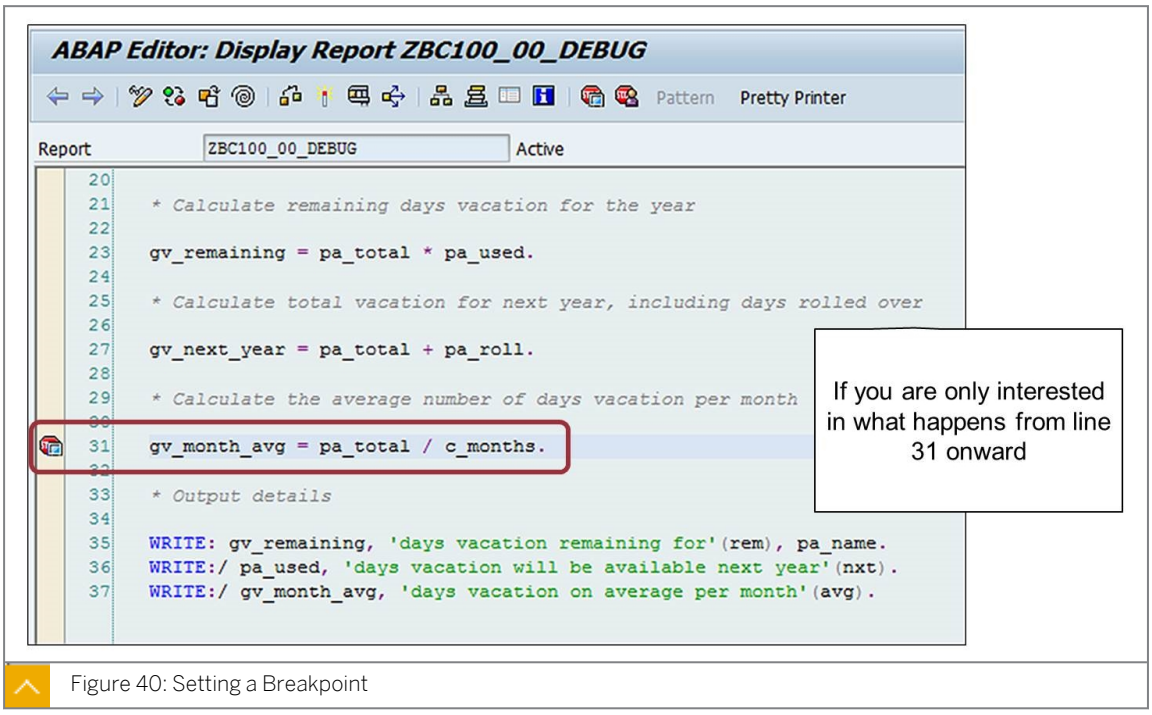

Figure 40: Setting a Breakpoint

Sometimes, you might only want to activate the Debugger once the program reaches a certain point. In this case, you can set a breakpoint at a specific line of code. When the program reaches this line, the Debugger starts.

There are two ways that you can set a breakpoint from the ABAP Editor:

- Click in the status column to the left of the line of code at which you want to create the breakpoint (see the figure Setting a Breakpoint).

  To remove this breakpoint, click it again.

- Place your cursor on the appropriate line of code and choose *Set/Delete Breakpoint* on the toolbar.

When you have set the breakpoint, choose *Direct Processing*. When the program reaches the breakpoint, the Debugger activates.

Note:
You can only set a breakpoint if the source code is active.

Activating the Debugger From a Screen

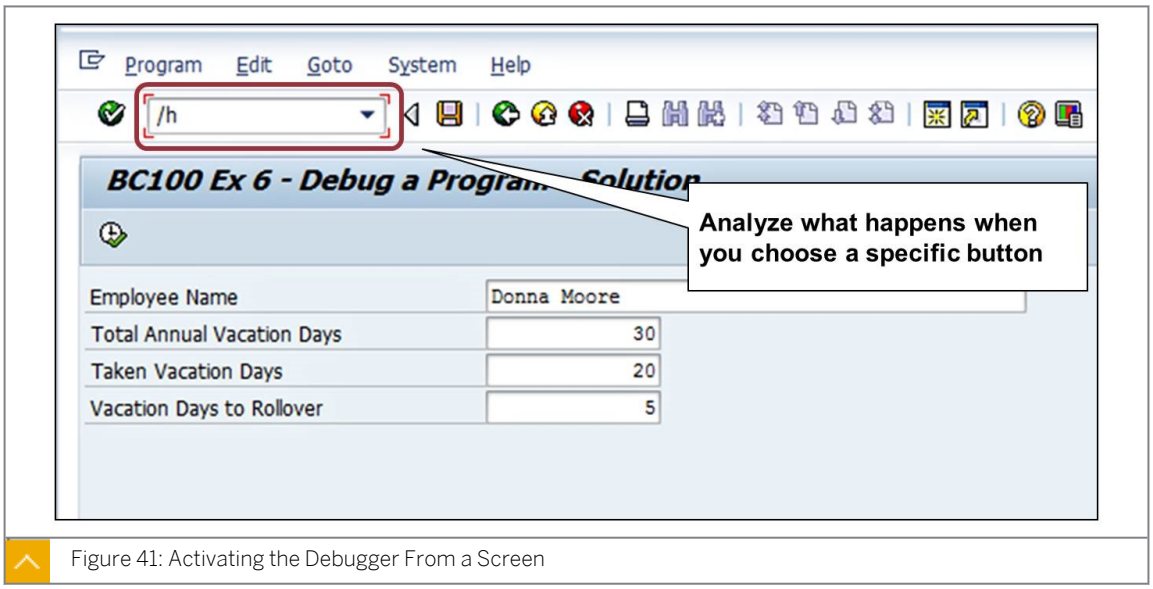

Figure 41: Activating the Debugger From a Screen

If your program has a screen, you can activate the Debugger from the screen (that is, you start debugging after the input fields are filled and the appropriate button or function is chosen).

First, start the program without the Debugger, fill any relevant input fields, and activate the Debugger before executing the function. To activate the Debugger from a screen, enter **`/h`** in the command field in the standard toolbar (see the figure Activating the Debugger from a Screen), then press **`ENTER`**.

The system displays a message to tell you that the Debugger has been activated, and the Debugger starts from the next user action (typically pressing a button to submit the data or execute the program).

Understanding the Debugger

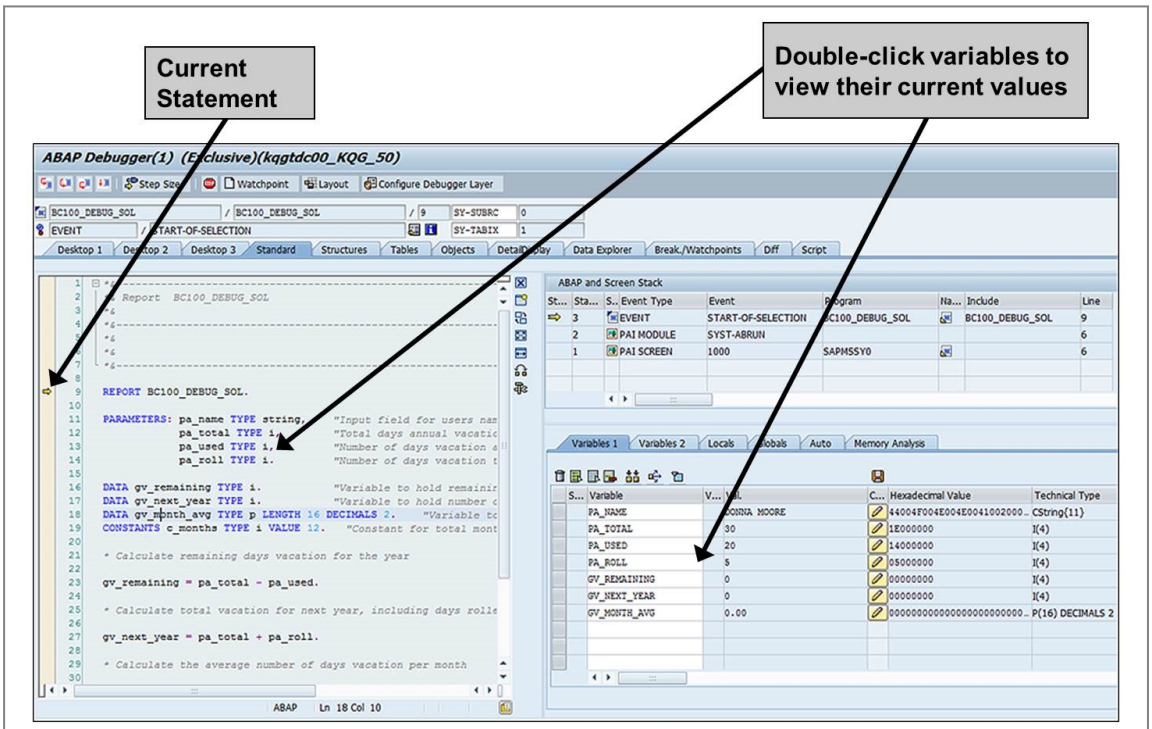

Figure 42: Understanding the Debugger

Once the Debugger starts, you can use it to analyze the code and determine how to solve any problems.

By default, the Debugger displays the source code in the screen area on the left (this is dependent on the release of the system, and the source code may display at the top of the screen instead). In the source code, a yellow arrow indicates the line of code that is to be executed next (see the figure Understanding the Debugger). This indicator allows you to track the program's progress.

You can also decide which variables you want to monitor. Double-click a variable in the source code to add it to the *Variables* screen area. This screen area displays the current value of any variables that you add. As the program progresses, you can see the values change as different statements are executed. The change in variable values can help you to identify mistakes in the code: by seeing when data changes, you can analyze the statements that cause those changes and identify any problems.

Moving Through Code in the Debugger

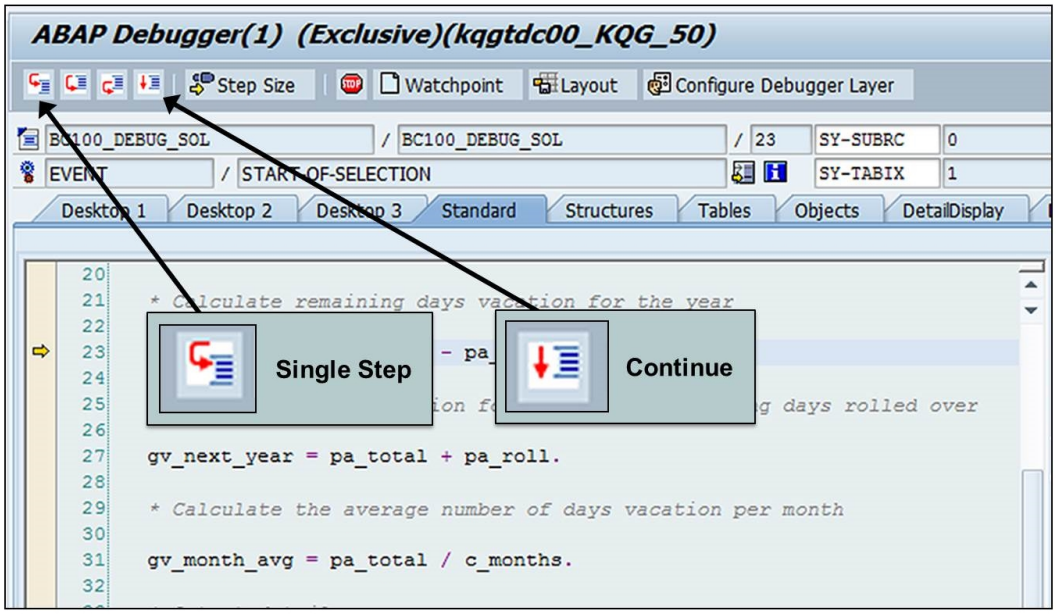

Figure 43: Moving Through Code in the Debugger

The toolbar in the Debugger contains some similar-looking buttons. These buttons are important because they allow you to 'walk through' the program. Two of the most commonly used buttons are the  (*Single Step*) and  (*Continue*) buttons.

The *Single Step* button steps through the source code one statement at a time. However, you do not have to step through every statement in a program. You may have seen enough to solve your problem or understand the program logic. In this case, choose the *Continue* button.

The *Continue* button behaves differently depending on the settings you have made. If you choose it, and there is a breakpoint later in the code, the Debugger continues to the next breakpoint and stops. If there are no breakpoints (or watchpoints, which are discussed later in this lesson), the Debugger continues to the end of the executable program and the output displays.

Different Types of Breakpoint

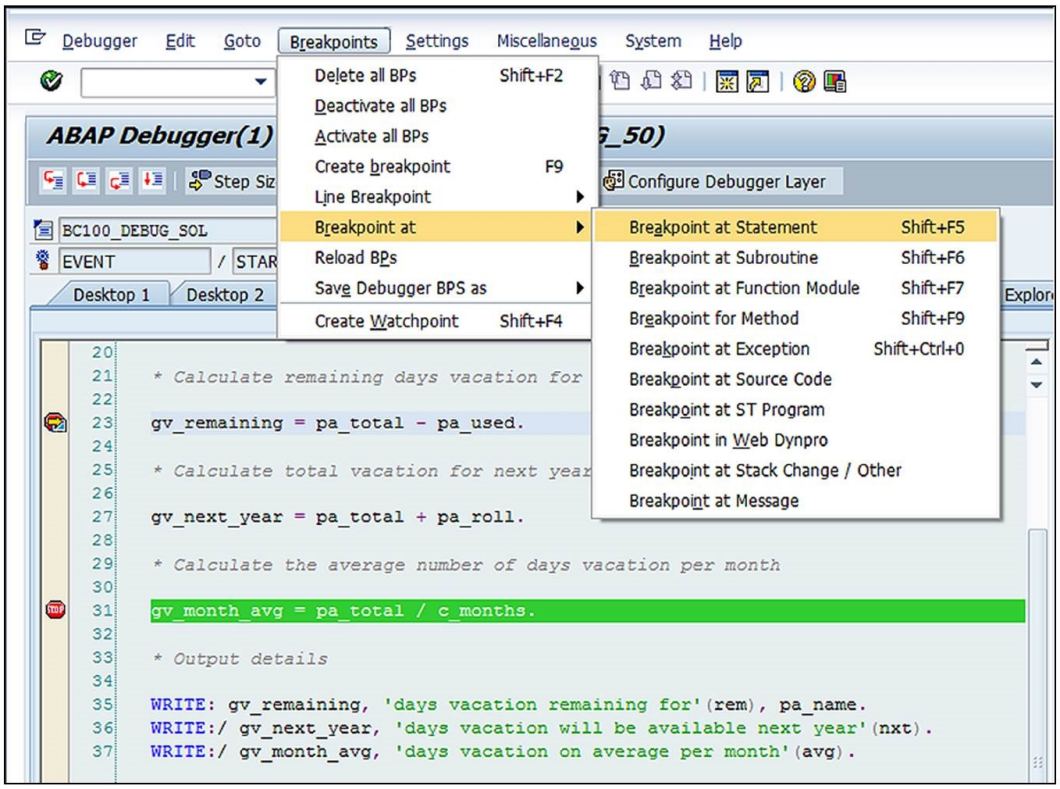

Figure 44: Different Types Of Breakpoint

Breakpoints allow you to mark lines in a program where you want the Debugger to stop. You know how to set a breakpoint in the ABAP Editor before executing the program, but you can also set a breakpoint while the Debugger is active. This type of breakpoint is known as a **Debugger breakpoint**.

As with breakpoints that are set in the Editor, you can set a Debugger breakpoint by clicking in the beige status column beside a line of code. By default, lines of code that are assigned a Debugger breakpoint are highlighted in green to differentiate them from breakpoints assigned in the Editor.

If you set additional breakpoints (either Editor or Debugger breakpoints), the *Continue* button causes the program to proceed until it reaches the next breakpoint. This means that breakpoints allow you to speed through sections of code that you do not need to analyze and focus on sections that require examination.

While in the Debugger, you can also set a breakpoint by choosing *Breakpoints → Breakpoint at*. This method provides several options for setting breakpoints. For example, a common scenario is to set breakpoints at a specific ABAP statement, such as the `CLEAR` command. With this option, the Debugger sets breakpoints at all instances of the `CLEAR` command in the source code. You can view a list of all breakpoints in your program by choosing the *Break./ Watchpoints* tab page. On this tab page you can select and delete breakpoints as required.

Watchpoints

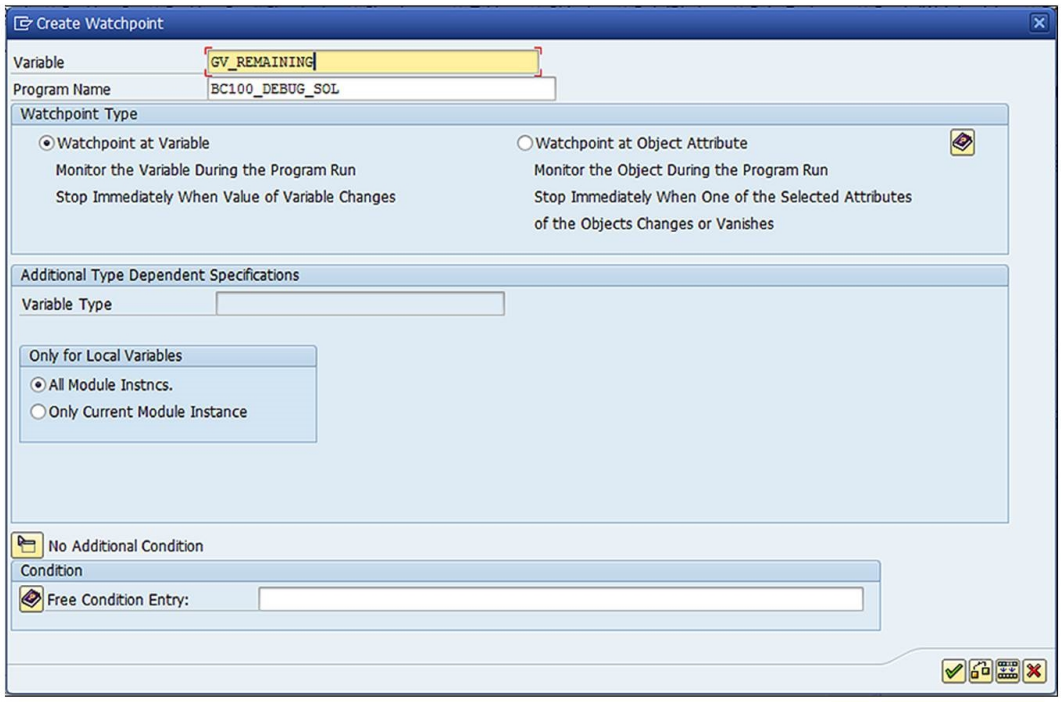

Figure 45: Watchpoints

Watchpoints are similar to breakpoints: a watchpoint can also be used to specify where the Debugger should stop. However, while breakpoints are set at either a specific line of code or an ABAP keyword, a watchpoint tells the Debugger to watch a variable and to stop the program when the contents of that variable change.

To set a watchpoint, you must be in the Debugger. In the toolbar of the Debugger, choose *Watchpoint*. The *Create Watchpoint* dialog box appears.

The easiest way to create a watchpoint is to enter the name of the variable you want to watch (see the figure Watchpoints), then choose the *Check* button. This ensures that you have entered the variable name correctly and that the Debugger recognizes the variable name. After this, choose **`Create and Close Window`**.

Watchpoints are useful when you are debugging a complex program, since the contents of a variable could change frequently. With watchpoints, you can avoid stepping through the code one line at a time. Instead, you can watch the variables that you are interested in and focus on statements that change those variables.

Note:
The Debugger stops **after** the contents of the variable change. To identify problems, analyze the code just above the yellow arrow when the Debugger reaches the watchpoint.

LESSON SUMMARY

You should now be able to:

- Debug a program

Unit 2
Lesson 6

Creating an ABAP List

LESSON OVERVIEW
In this lesson, you learn how to create an ABAP list and change how the list is formatted.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Create an ABAP list

ABAP List Functionality

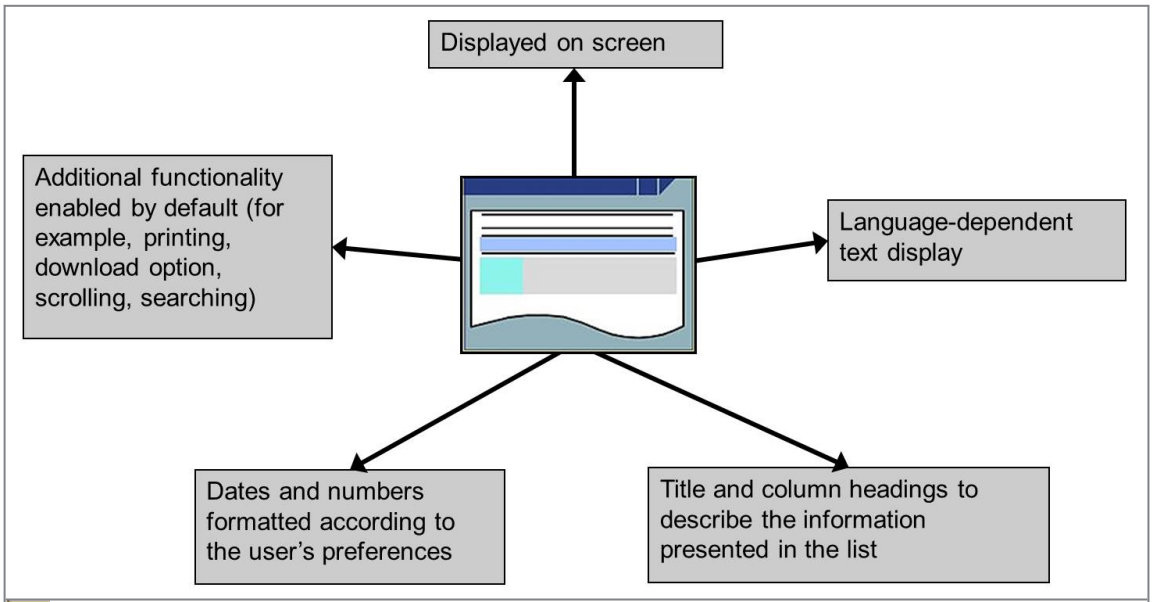

Figure 46: List Functionality

The `WRITE` statement is one of the first keywords used in this course. You know that the `WRITE` keyword can be used to output text and information to the screen, creating an ABAP list. Therefore, you can create simple reports by using the `WRITE` statement to output information that the user needs to see.

You also saw that lists can be designed for a number of languages by using text symbols to translate the texts output by the `WRITE` statement. With text symbols, the logon language of the user determines which translation is output by the program.

Standard List Functionality

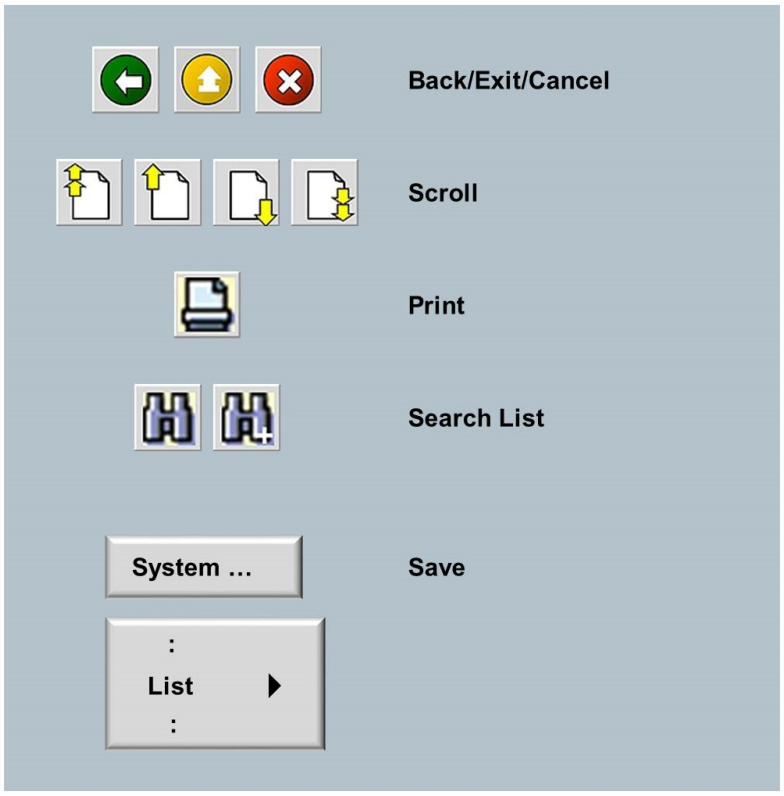

Figure 47: Standard List Functionality

Program users may also have further requirements for ABAP programs that generate lists. For example, users might require the following features:

- A print function
- A ‘save as’ function

  For example, the ability to save a copy of the list as a text file or spreadsheet.

Fortunately, you do not need to explicitly program these features; they are available by default from any ABAP list. Once the user knows how to use these functions in the standard SAP environment, you do not need to write any additional code.

Use of Colors in an ABAP List

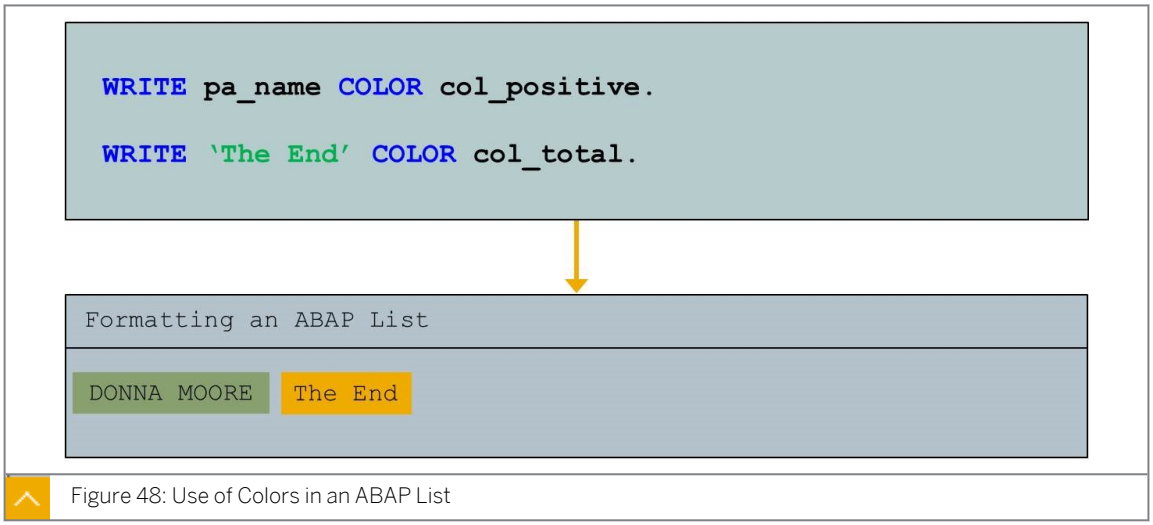

Figure 48: Use of Colors in an ABAP List

You can also improve the presentation of an ABAP list. Some of the ways you can format a list include the following:

- Use of colors and icons
- Use of list titles and column headings

The figure Use of Colors in an ABAP List shows some of the colors that you can apply. These colors are easy for a programmer to implement, and require only minor development effort. The full list of colors, and the required syntax, is shown in the table ABAP List Colors.

Table 6: ABAP List Colors

Syntax for Color	Color
COL_BACKGROUND	GUI-dependent
COL_HEADING	Gray-blue
COL_NORMAL	Light gray
COL_TOTAL	Yellow
COL_KEY	Blue-green
COL_POSITIVE	Green
COL_NEGATIVE	Red
COL_GROUP	Orange

Use of Icons in an ABAP List

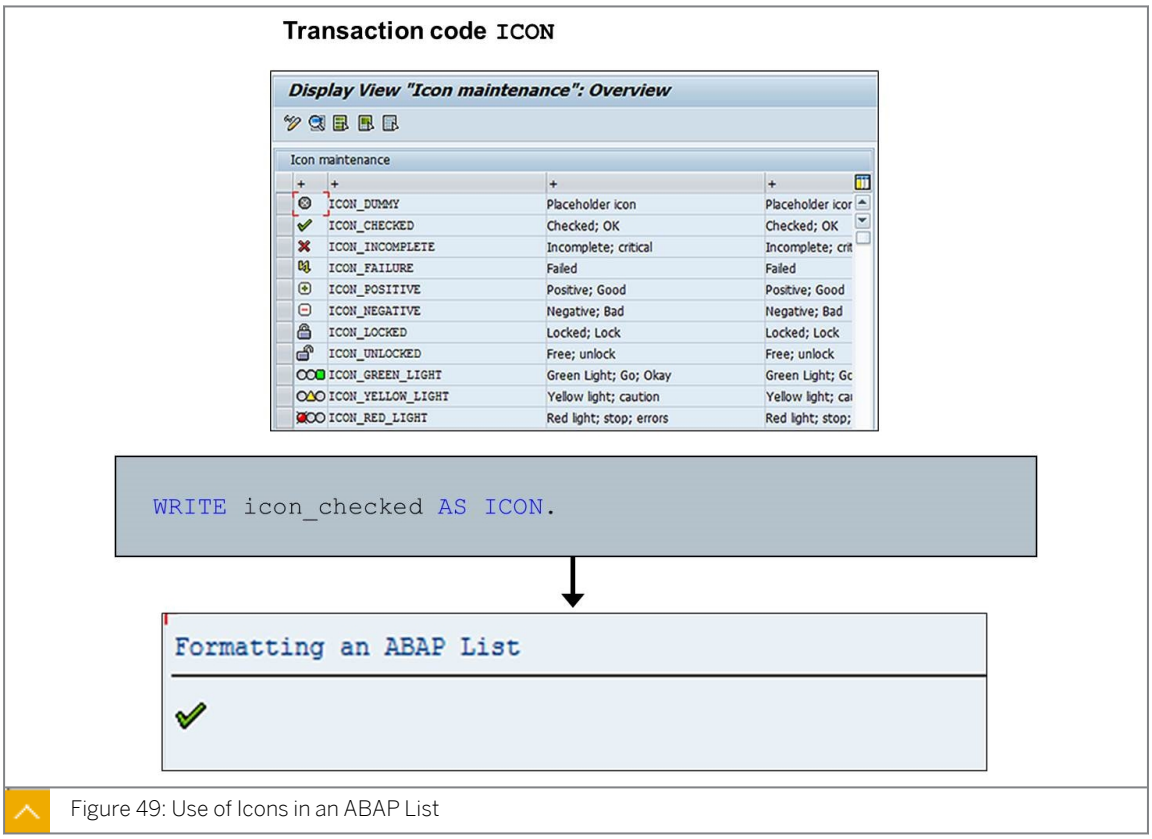

Figure 49: Use of Icons in an ABAP List

You can display icons in an ABAP List by using the addition `AS ICON` with a `WRITE` statement. You can find a full list of icons on the *Icon Maintenance* screen (transaction code `ICON`). The figure Use of Icons in an ABAP List demonstrates the *Icon Maintenance* screen, the syntax to display an icon, and an example of an icon in the output.

ABAP List Header

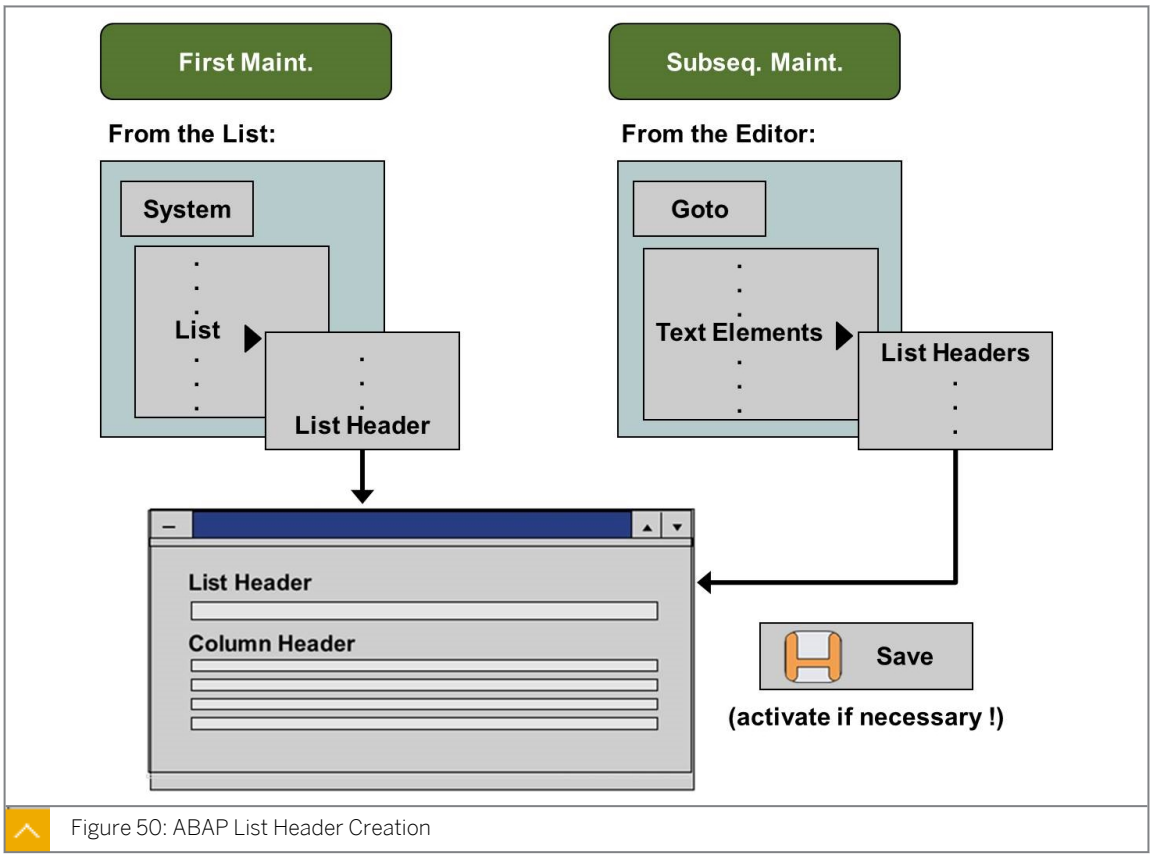

Figure 50: ABAP List Header Creation

In an executable program (or report), you can maintain a single line **list header** and a maximum of four rows of **column headers**. If you do not maintain a list header, the title of the program (taken from the program attributes) is used as the list header.

List headers and column headers can be useful when the report displays a list of records. Column headers make it easy for users to understand the data.

There are two ways to maintain these headers (as shown in the figure ABAP List Header Creation). Both methods require the program to be active.

The first method is to execute the program, then maintain the headers when the output displays. You can do this by choosing *System* → *List* → *List Header* on the output screen.

The second method is to access the maintenance screen for changing headers from the ABAP Editor. You can do this by choosing *Goto* → *Text Elements* → *List Headings*. With this method, it is more difficult to align column headings with the data in the list. However, from this screen, you can translate the list and column headers by choosing *Goto* → *Translation*.

List with Column Headers

The figure Example List with Column Headers shows an example of a list with column headers to describe the presented data.

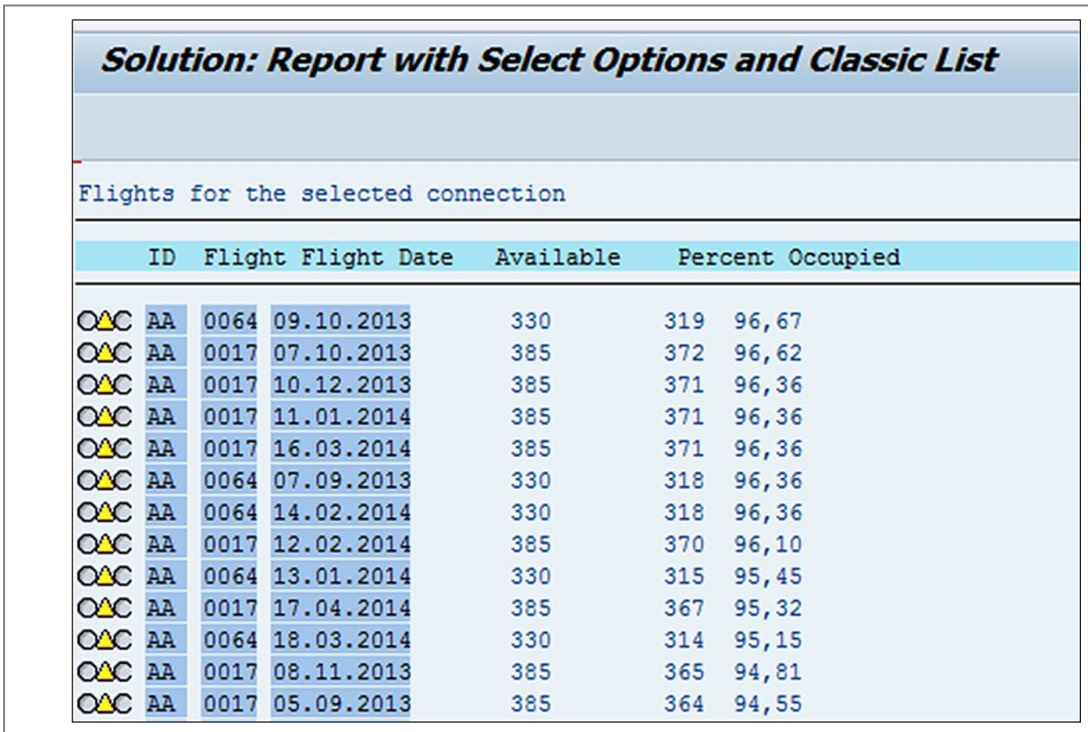

Figure 51: Example List with Column Headers

Note:
You can use these methods to create headers for ABAP lists that have been generated using the `WRITE` statement. The course ABAP Workbench Foundations (BC400) discusses the ABAP List Viewer (or ALV Grid), with which you can present multiple, identically structured records to the user. When using the ABAP List Viewer, you can derive column headings automatically from the ABAP Dictionary.

LESSON SUMMARY
You should now be able to:

- Create an ABAP list

Unit 2
Lesson 7

Processing Character Strings

LESSON OVERVIEW
In this lesson, you learn how to process character strings in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Process character strings in a program

Character String Functions

So far, this course has focused on working with numeric data. This lesson discusses some of the different ways in which character-type data can be processed and manipulated.

The following are some of the character-like data types that will be discussed:

- `string`
- `c`
- `d`
- `t`

ABAP Program

```
PARAMETERS: pa_fname TYPE c LENGTH 30,
            pa_lname TYPE c LENGTH 30.

DATA gv_full_name TYPE string.

CONSTANTS gc_comma TYPE c LENGTH 1 VALUE ','.

CONCATENATE pa_fname pa_lname INTO gv_full_name
  SEPARATED BY space.

* Different separators can be used

CONCATENATE pa_fname pa_lname INTO gv_full_name
  SEPARATED BY gc_comma.
```

Figure 52: Concatenating Data

You can use the `CONCATENATE` statement to create one character string from two separate character strings. The `SEPARATED BY` addition allows you to insert a separator of your choice between the two variables.

If the target variable is shorter than the combined length of the source values, the length of the new character string is restricted to the length of the target field. Therefore, we recommend that you use the `string` data type (which has an unspecified length) for the target variable when concatenating.

Searching for a Value in a String

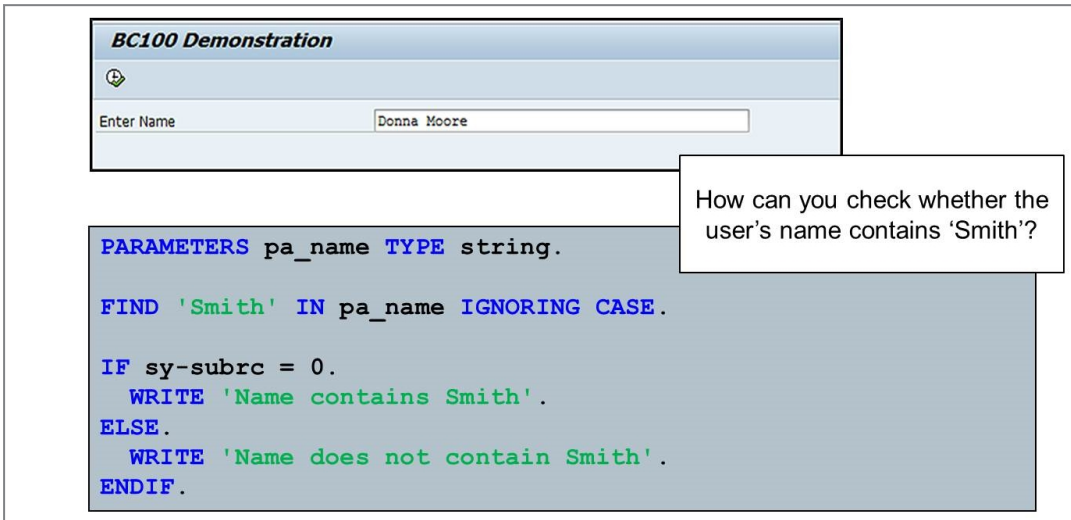

Figure 53: Searching for a Value in a String

The `FIND` and `REPLACE` statements are important for processing character strings. You can use `FIND` to search for a value in a character string, and `REPLACE` to replace a value. The figure Searching for a Value in a String demonstrates an example of the `FIND` statement. The following code demonstrates an example of the `REPLACE` statement.

```
PARAMETERS pa_name TYPE string.

REPLACE 'Smith' IN pa_name WITH 'Jones' IGNORING CASE.

IF sy-subrc = 0.

  WRITE: 'New name is:', pa_name.

ENDIF.
```

When the program processes these statements, it sets the system variable `sy-subrc`. If this variable is set to 0, it indicates that the statement was successful. In the `FIND` statement, this means that the value or pattern was found in the variable. In the `REPLACE` statement, it means that the value was found and successfully replaced with the new value.

The `FIND` and `REPLACE` statements are based on similar principles. While `FIND` simply searches for the sub-sequence or pattern in a variable, `REPLACE` searches for the pattern and replaces any occurrences with the supplied content.

Note:
For more information about the `FIND` and `REPLACE` keywords, see the ABAP keyword documentation.

The SPLIT Keyword

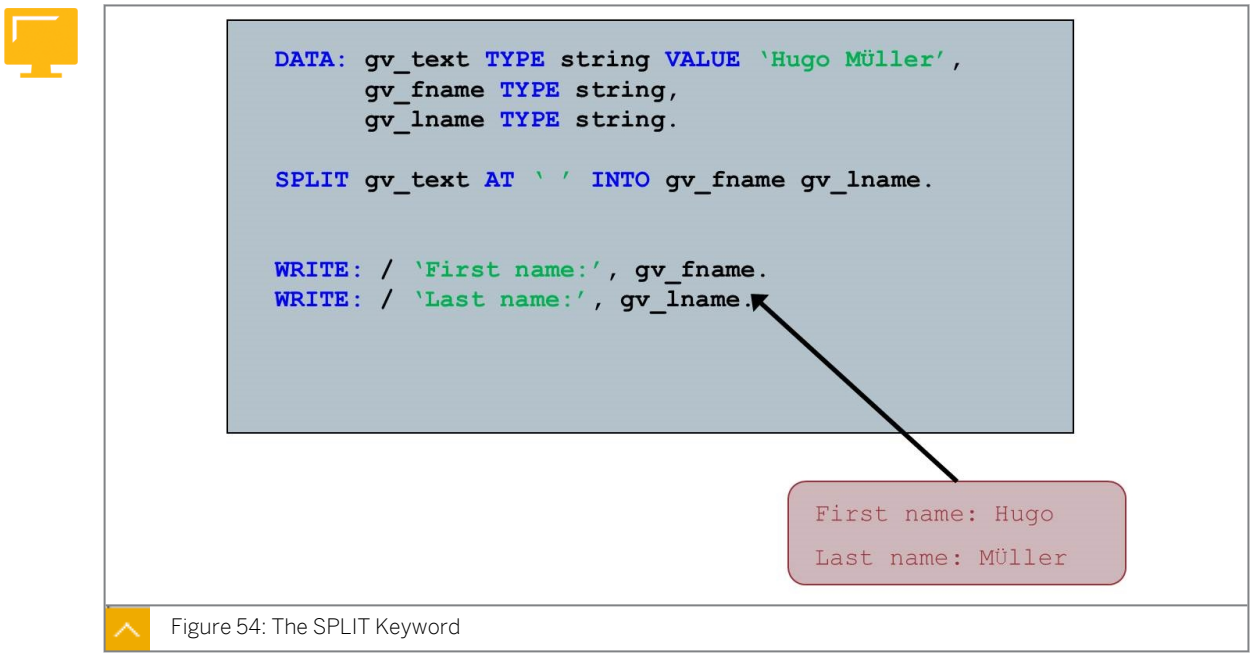

Figure 54: The SPLIT Keyword

The figure, The Split Keyword, shows an example of the `SPLIT` keyword. Using this statement, you can divide character strings into substrings. The character string is split before and after the separator that you specify (in the figure, a space) and is then divided into individual target variables.

If you do not supply enough target variables, the remainder of the source character string (including the separator) is entered into the last target variable.

LESSON SUMMARY

You should now be able to:

- Process character strings in a program

Learning Assessment

1. Which of the following are numeric data types?

   Choose the correct answers.

   - ☐ A `string`
   - ☐ B `i`
   - ☐ C `c`
   - ☐ D `p`
   - ☐ E `t`

2. Variables are declared using the `DATA` keyword, and either an ABAP standard type or a global type from the ABAP Dictionary can be used to describe them.

   Determine whether this statement is true or false.

   - ☐ True
   - ☐ False

3. Which of the following items is supplied in the logon language of the user instead of the hard-coded text in the ABAP program when a statement using the item is executed?

   Choose the correct answer.

   - ☐ A Literal
   - ☐ B Text symbol
   - ☐ C User's name
   - ☐ D String data type

4. Which of the following are valid operators that can be used in arithmetic operations in ABAP?

   Choose the correct answers.

   - ☐ A %
   - ☐ B /
   - ☐ C !
   - ☐ D x
   - ☐ E *

5. When working with system variables such as `SY-DATUM` or `SY-UZEIT`, you must populate the variables with the appropriate values.

   Determine whether this statement is true or false.

   - ☐ True
   - ☐ False

6. Which command can you enter in the command field to start the ABAP Debugger when a button is chosen on a screen?

   Choose the correct answer.

   - ☐ A **`/x`**
   - ☐ B **`/h`**
   - ☐ C **`/n`**
   - ☐ D **`/o`**

7. Which of the following features can be used in an ABAP list without requiring the developer to program those functions?

   Choose the correct answers.

   - ☐ A Print
   - ☐ B Add icons to output
   - ☐ C Add color to values in output
   - ☐ D Search
   - ☐ E Scroll

8. You can use one of the following statements to divide a character string into several substrings.

   Choose the correct answer.

   ☐ A `CONCATENATE`

   ☐ B `WRITE`

   ☐ C `REPLACE`

   ☐ D `SPLIT`

Learning Assessment - Answers

1. Which of the following are numeric data types?

   Choose the correct answers.

   - [ ] A `string`
   - [X] B `i`
   - [ ] C `c`
   - [X] D `p`
   - [ ] E `t`

   You are correct! ABAP Standard Data Types can be grouped in Character-like and Numeric Data Types. Numeric Standard Data Types are i (integers, whole numbers and no decimal places) and p (packed, decimal numbers). For more information, see Unit 2, Lesson 1 Defining Simple Variables, task ABAP Standard Data Types.

2. Variables are declared using the `DATA` keyword, and either an ABAP standard type or a global type from the ABAP Dictionary can be used to describe them.

   Determine whether this statement is true or false.

   - [X] True
   - [ ] False

   You are correct! You can define a variable using the `DATA` keyword. To provide a description or type for the variable, you can use either an ABAP standard type or a global type from the ABAP Dictionary. For more information, see Unit 2 Lesson 1 Defining Simple Variables, task Variable Declarations.

3. Which of the following items is supplied in the logon language of the user instead of the hard-coded text in the ABAP program when a statement using the item is executed?

   Choose the correct answer.

   - [ ] A Literal
   - [x] B Text symbol
   - [ ] C User's name
   - [ ] D String data type

   You are correct! Since text literals (hard-coded texts) cannot be changed, you cannot translate them into other languages. The solution is to use text symbols instead of text literals. If you are developing productive programs that will be executed by different users in different logon languages, use text symbols to ensure that all texts are translatable. For more information, see Unit, Lesson 2: Defining Text Symbols, task Text Symbols.

4. Which of the following are valid operators that can be used in arithmetic operations in ABAP?

   Choose the correct answers.

   - [ ] A %
   - [x] B /
   - [ ] C !
   - [ ] D x
   - [x] E *

   You are correct! ABAP allows you to program arithmetic operations easily. The list of valid operators for an arithmetic operation includes the following: + (Addition), - (Subtraction), * (Multiplication), / (Division). For more information, see Unit 2, Lesson 3: Performing Arithmetic Operations Using Simple Variables, task Arithmetic Operators for ABAP Calculations.

5. When working with system variables such as `SY-DATUM` or `SY-UZEIT`, you must populate the variables with the appropriate values.

   Determine whether this statement is true or false.

   - [ ] True
   - [x] False

   That is correct! System variables provide information about the actual system status. The ABAP runtime system populates and changes the values of the system fields when necessary. For example, the system variable `SY-DATUM` is always filled with the current date, and `SY-UZEIT` is filled with the current time. For more information, see Unit 2, Lesson 4: Using System Variables, task System Variables.

6. Which command can you enter in the command field to start the ABAP Debugger when a button is chosen on a screen?

   Choose the correct answer.

   ☐ A **/x**

   ☒ B **/h**

   ☐ C **/n**

   ☐ D **/o**

   You are correct! To activate the Debugger from a screen, enter **/h** in the command field in the standard toolbar, then press ENTER. For more information, see Unit 2, Lesson 5: Debugging a Program, task Activating the Debugger From a Screen.

7. Which of the following features can be used in an ABAP list without requiring the developer to program those functions?

   Choose the correct answers.

   ☒ A Print

   ☐ B Add icons to output

   ☐ C Add color to values in output

   ☒ D Search

   ☒ E Scroll

   You are correct! Scroll, Print, Search, Save as, are features available by default for any ABAP list. Therefore, you do not need to explicitly program these features. For more information, see Unit 2, Lesson 6: Creating an ABAP List, task Standard List Functionality.

8. You can use one of the following statements to divide a character string into several substrings.

   Choose the correct answer.

   ☐ A `CONCATENATE`

   ☐ B `WRITE`

   ☐ C `REPLACE`

   ☒ D `SPLIT`

   You are correct! Using the `SPLIT` statement, you can divide character strings into substrings. The character string is split before and after the separator that you specify and is then divided into individual target variables. For more information, see Unit 2, Lesson 7: Processing Character Strings, task The SPLIT Keyword.

UNIT 3 Flow Control Structures in ABAP

UNIT OBJECTIVES

- Implement conditional logic in a program
- Implement loops in a program

Implementing Conditional Logic

LESSON OVERVIEW
This lesson introduces conditional logic and shows you how to use it in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Implement conditional logic in a program

Conditional Branches

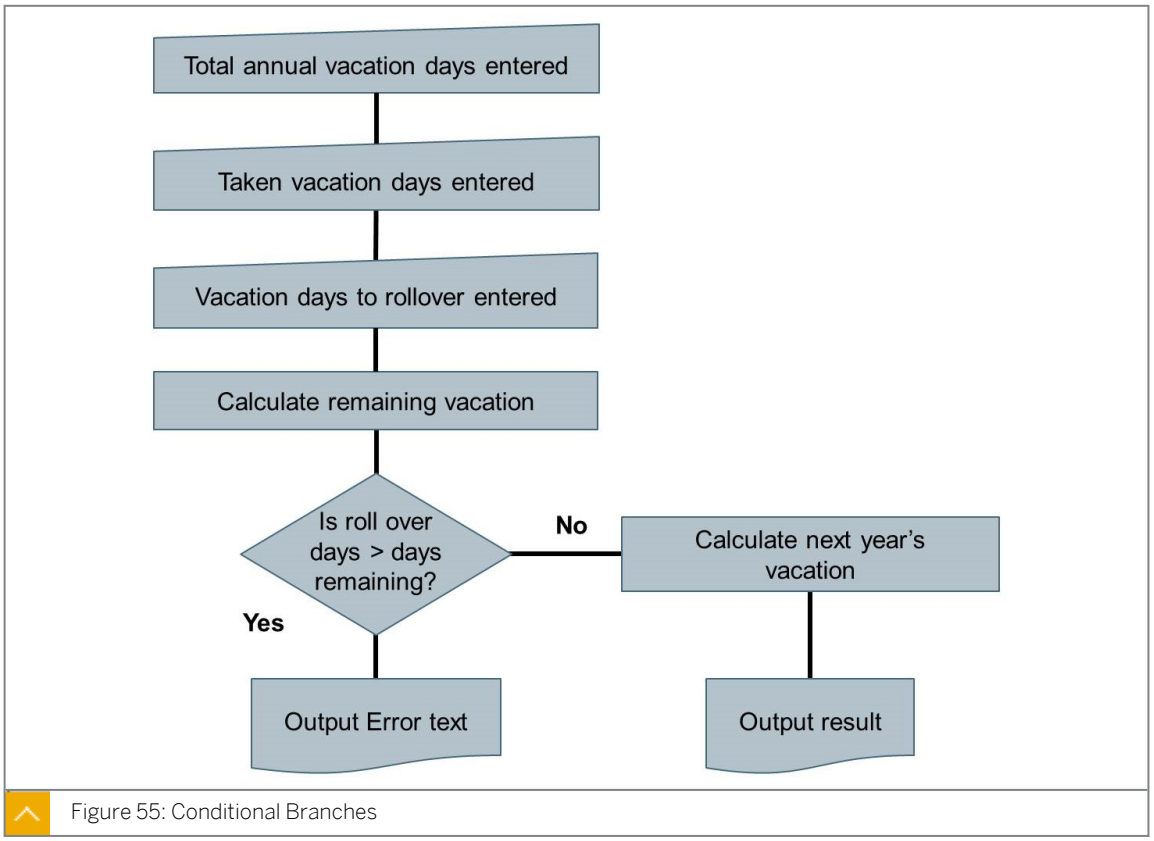

Figure 55: Conditional Branches

When designing an ABAP program, you will frequently encounter situations where different sequences of statements need to be executed depending on certain conditions. In these situations, it can be helpful to create a program flowchart to describe how the program should work.

Most programs can be broken down into the following components:

- Sequences

One or more operations that are to be performed.

- Branches

  Questions with a yes or no answer.

- Case distinctions

  Questions with multiple possible answers.

- Loops

  A repeating sequence.

The figure Conditional Branches shows a program flowchart that includes a conditional branch.

Program Flowchart

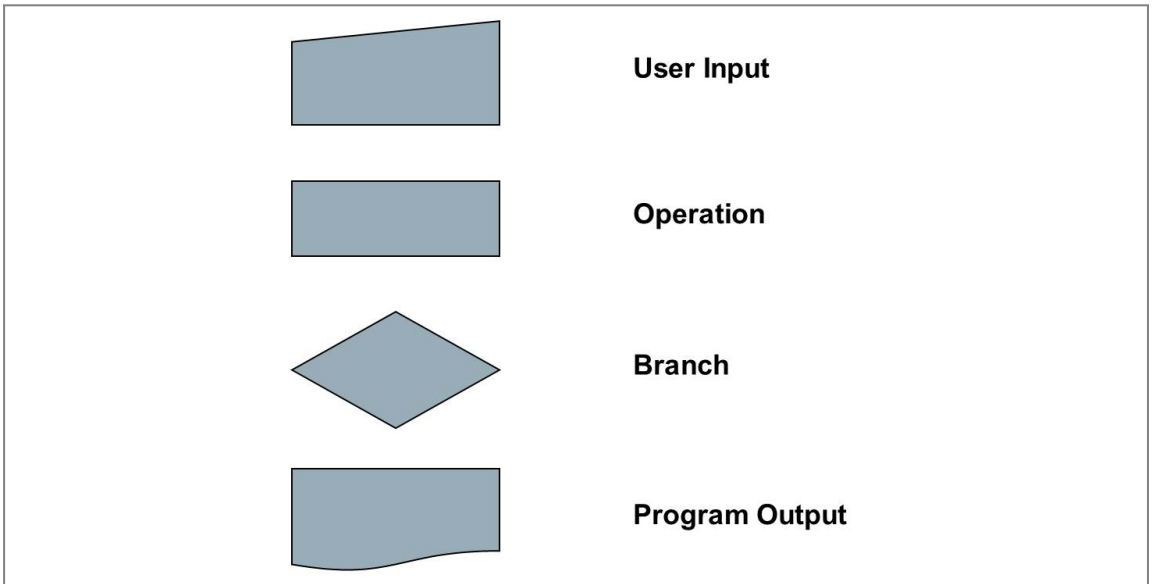

Figure 56: Symbols Used in Program Flowcharts

A program flowchart is a flow diagram that can be used as a starting point for designing your program. It displays the program flow visually. You can use program flowcharts for all programming languages, not just ABAP. In a flowchart, symbols are used to represent different actions, inputs, and processing. The figure, Symbols Used in Program Flowcharts, illustrates some of these symbols.

Implement a Branch

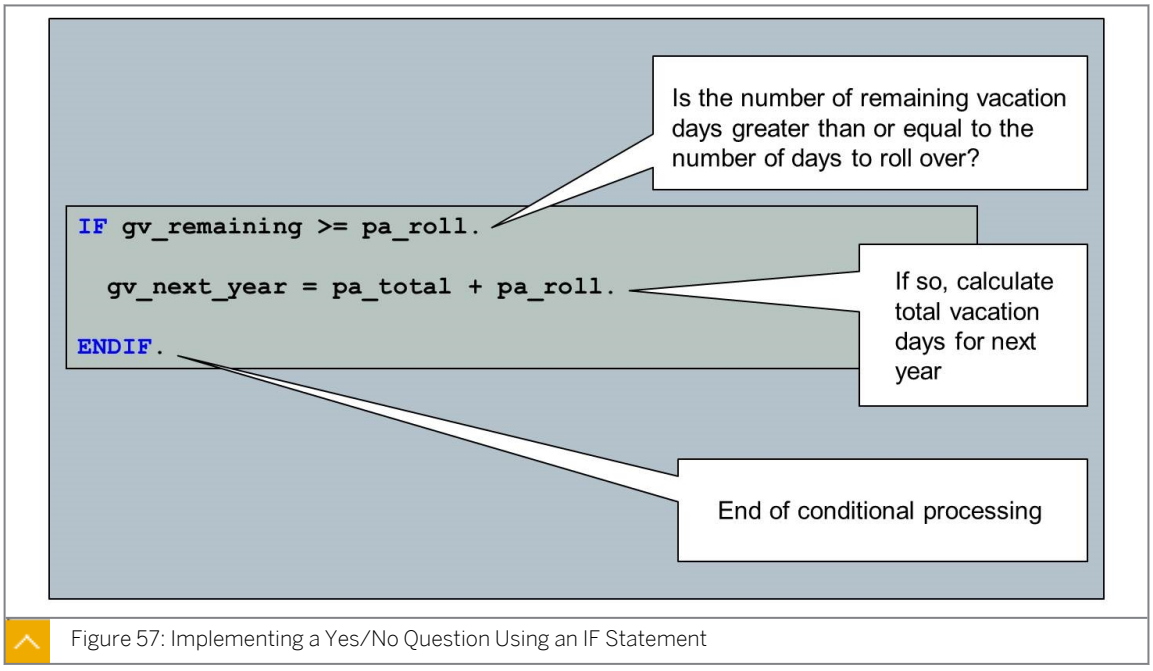

Figure 57: Implementing a Yes/No Question Using an IF Statement

You implement a branch (that is, a question with a yes or no answer) in your code by using a logical expression. In the simplest scenario, this means a comparison between a target value and an actual value.

The result of the expression is either true or false. For example, in the figure Implementing a Yes/No Question Using an `IF` Statement, the expression compares the current value of the variable `gv_remaining` to the value of the selection screen parameter `pa_roll`. If `gv_remaining` is greater than or equal to `pa_roll` (that is, if the number of remaining vacation days is greater than or equal to the number of days to be rolled over), this expression evaluates as **true**.

Since the condition is true, the program proceeds to add the rolled over vacation days to next year's allowance.

If the Condition is False

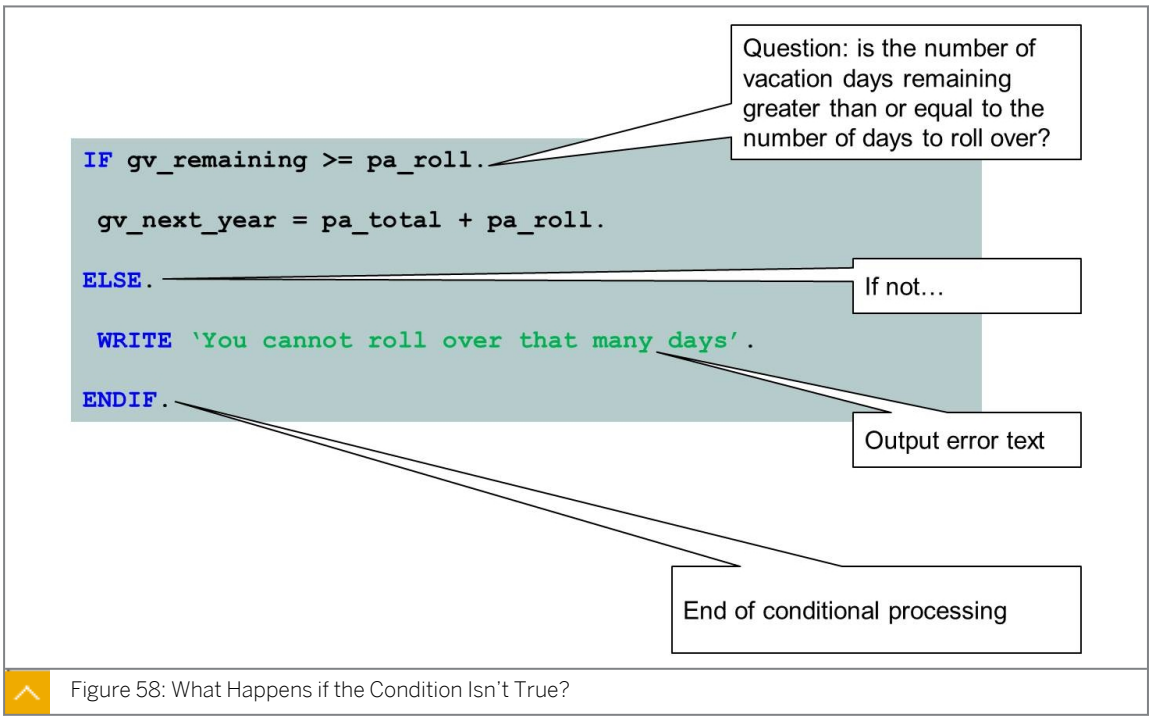

Figure 58: What Happens if the Condition Isn't True?

You must also consider how your program proceeds if the condition is false. In the figure, What Happens if the Condition Isn't True, the program outputs a simple error text when `gv_remaining` is not greater than or equal to `pa_roll`.

IF Statements

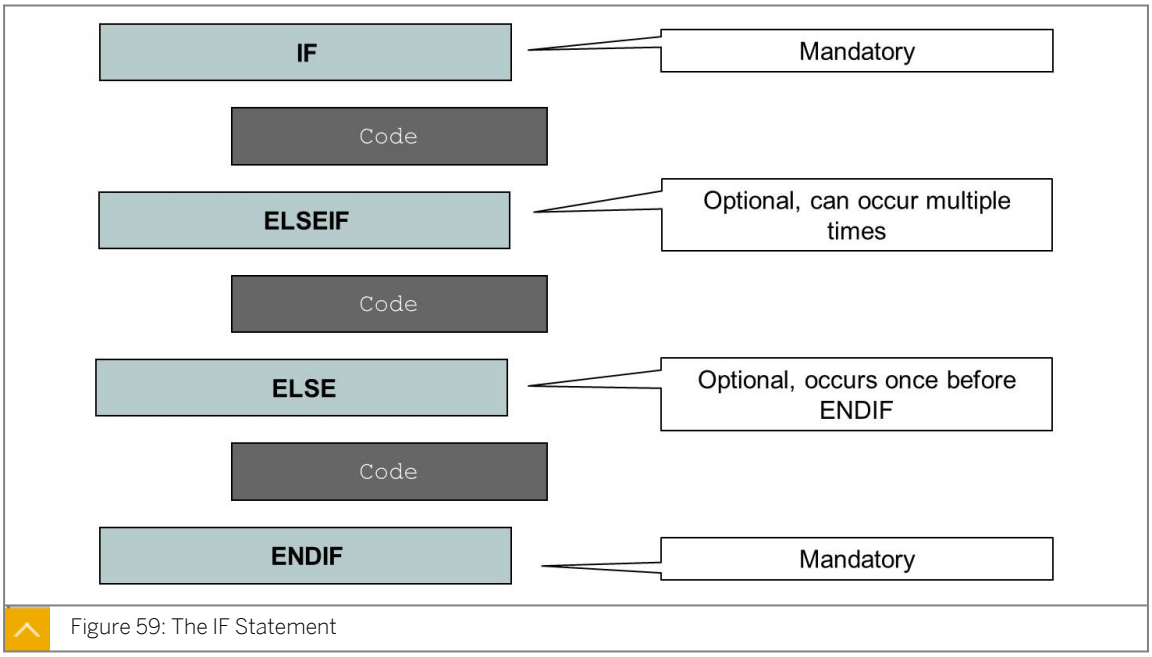

Figure 59: The IF Statement

In ABAP, you can use the **`IF` construct** to execute different sequences of statements depending on certain conditions.

With an IF construct, you can define **any logical expressions** as check conditions. If the IF condition is true, the relevant block of code is executed. Otherwise, the condition specified in the next ELSEIF branch is checked. You can define several ELSEIF branches, and the program checks them in sequence until a branch evaluates as true or all branches have been checked. If the condition in one branch is true, the statement block is executed and the program jumps to the ENDIF. If none of the specified conditions are met, the system executes the ELSE branch (if you have defined it).

ELSEIF and ELSE branches are not mandatory, but every IF must have an ENDIF.

Note:
You can also implement conditional logic using the CASE statement. The CASE statement is covered in more detail in the course ABAP Workbench Foundations (BC400).

CASE Statement

You can use the CASE statement to clearly distinguish cases. The content of the field or variable specified in the CASE part is checked against the variables or values listed in the WHEN branches. If the field contents match, the program processes the respective statement block. If none of the comparisons match, the program processes the WHEN OTHERS branch (if you have defined it). The first WHEN branch is mandatory, but all other additions are optional. The following code gives an example:

```
CASE gv_var1.
 WHEN 'A'.

    "Do something

 WHEN 'B'.

    "Do something else

 WHEN 'C'.

    "Do something different again

 WHEN OTHERS.

    "Logic if none of the other branches are executed

ENDCASE.
```

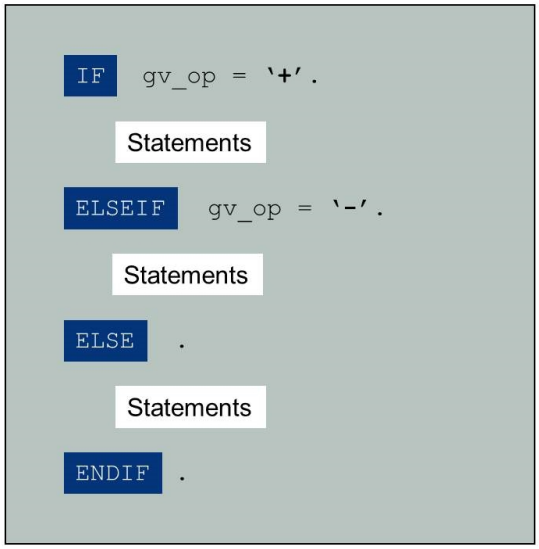

Figure 60: The IF Statement

The conditions of an `IF` statement can be simple or complex. For example, you might need to consider the values of more than one variable. To do this, you can use the `AND` or `OR` operators in the condition.

AND or OR in an IF Statement

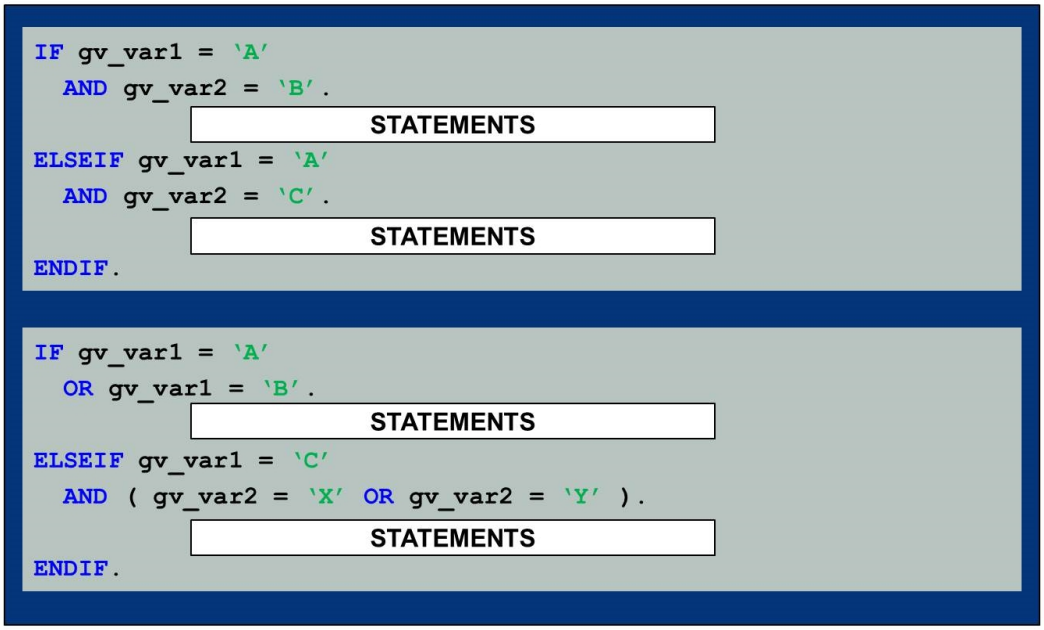

Figure 61: Use of AND or OR in an IF Statement

When you join multiple logical expressions with `AND`, a new logical expression is formed. This logical expression is only true when **all** logical expressions that comprise it are true. If one of the logical expressions is false, the operation is also false.

In the first example in the figure Use of `AND` or `OR` in an `IF` statement, the first branch is processed only if `gv_var1` has the value 'A' and `gv_var2` has the value 'B'.

When you join multiple logical expressions with `OR`, another logical expression is formed. This logical expression is true if **at least one** of the logical expressions that comprise it are true. If all of the logical expressions are false, the operation evaluates as false.

In the second example in the figure, the first branch processes if `gv_var1` has the value 'A' or `gv_var2` has the value 'B'. It is not necessary for both of these variables to have the value specified.

The `ELSEIF` branch of the second example demonstrates a condition where both an `AND` and an `OR` are combined. Note the use of the parentheses around the second logical expression (which evaluates whether `gv_var2` has the value 'X' or 'Y').

Relational and Boolean Operators

Table 7: Relational Operators

Operator	Abbreviation	Meaning
=	EQ	Equal
<>	NE	Not equal
>	GT	Greater than
>=	GE	Greater than or equal to
<	LT	Less than
<=	LE	Less than or equal to

The table ,Relational Operators, lists the relational operators that you can use to compare two variable or data objects. These operators can be used with variables of any data type.

Each operator has an equivalent abbreviation that behaves identically. In the figure, Examples of Relational Operators, both examples behave the same.

```
* Example 1

IF gv_var1 > 5.          " Use of operator
  ......
ENDIF.

* Example 2

IF gv_var1 GT 5.          " Use of abbreviation
  ......
ENDIF.
```

Figure 62: Examples of Relational Operators

Note:
We recommend that you use only one type of operator within the same program (that is, all operators or all abbreviations, but not a mix of both).

Boolean Operators

Table 8: Boolean Operators

Operator	Meaning
`AND`	Used to join multiple logical expressions to create a new logical expression that is true only when all of the multiple logical expressions are true.
`OR`	Used to join multiple logical expressions to create a new logical expression which is true if at least one of the logical expressions is true.
`NOT`	The negation of a logical expression using NOT creates a new logical expression that is false if the logical expression is true and vice versa.

Boolean operators either join (`AND` and `OR`) or negate (`NOT`) a logical expression.

LESSON SUMMARY

You should now be able to:

- Implement conditional logic in a program

Unit 3
Lesson 2

Implementing Loops

LESSON OVERVIEW
In this lesson, you learn about loops and how to use them in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Implement loops in a program

Loops

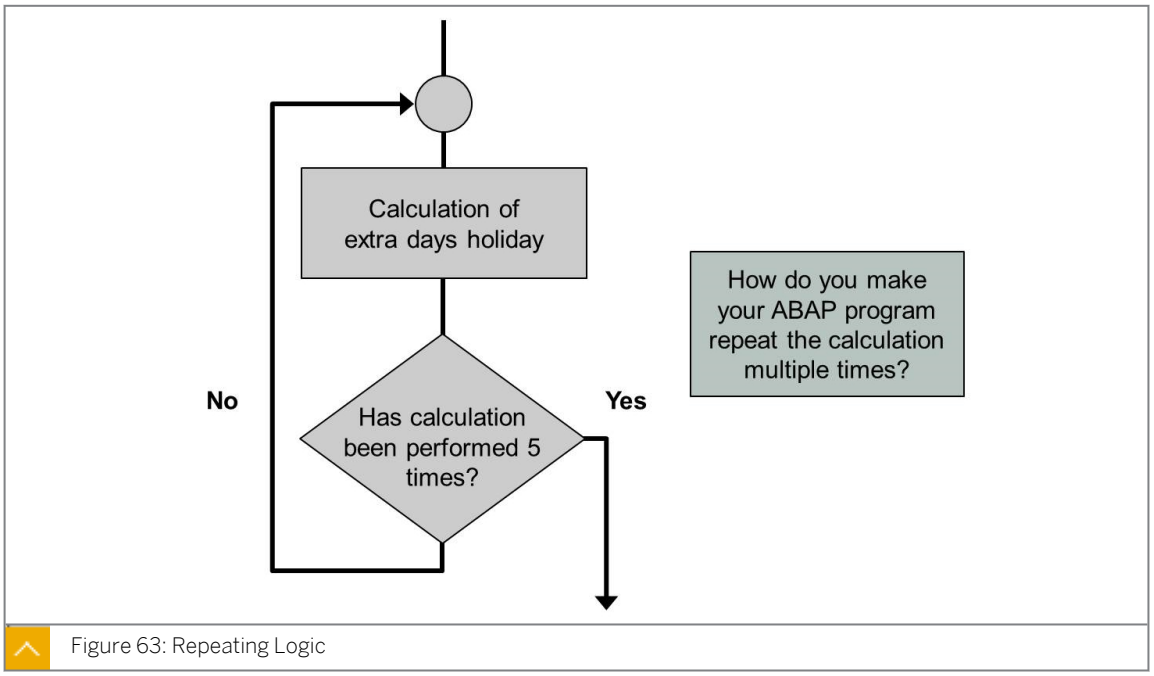

Figure 63: Repeating Logic

There are many situations in which you might want an ABAP program to process the same piece of logic several times. For example, you may want the program to repeat a calculation until a certain condition is met.

To implement this in your ABAP code, you can program a loop. A loop repeats an operation (or operations) until a predefined condition is met. The figure Repeating Logic illustrates the logic behind a loop.

Loop Constructs

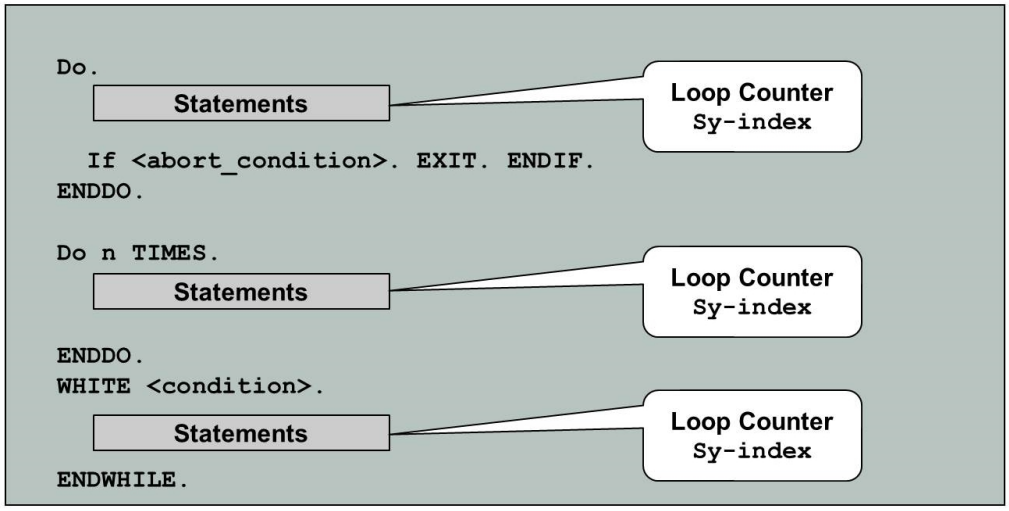

Figure 64: Loop Constructs

There are several loop constructs in ABAP. The figure, Loop Constructs, shows some of these constructs.

In `DO` and `WHILE` loops, the system variable `sy-index` contains the number of the current loop pass. Because of this, you do not need to maintain your own iteration counter within the code.

Example of a DO Loop

```
PARAMETERS: pa_name TYPE string,   "Input field for user's name
            pa_start TYPE i,       "Total days annual vacation when joining
            pa_years TYPE i.       "Years to calculate vacation for

DATA gv_new_tot TYPE i.            "Total annual vacation days in X years time

* Calculate total annual vacation in X years time
gv_new_tot = pa_start.             "First move current total to new variable

DO pa_years TIMES.

* Increment the total - extra days is the same as number of years of service
  gv_new_tot = gv_new_tot + sy-index.

* Extra days vacation given only up to and including the 5th year
  IF sy-index = 5.
    EXIT.
  ENDIF.

ENDDO.
```

Figure 65: Example of a DO Loop

The figure, Example of a `DO` Loop, demonstrates the source code for a `DO` loop. In this figure, the code within the loop will be processed repeatedly. The number of times it will be processed is determined by the number that the user enters in the `pa_years` input field.

However, the logic should only be repeated a maximum of five times. This is achieved by using the `EXIT` command in an appropriate `IF` statement.

LESSON SUMMARY

You should now be able to:

- Implement loops in a program

Learning Assessment

1. Which of the following are mandatory parts of an `IF` statement?

   Choose the correct answers.

   - ☐ A `IF`
   - ☐ B `ELSEIF`
   - ☐ C `ELSE`
   - ☐ D `ENDIF`

2. Which of the following system variables contains the current loop pass in a DO loop?

   Choose the correct answer.

   - ☐ A `SY-DATUM`
   - ☐ B `SY-UNAME`
   - ☐ C `SY-UZEIT`
   - ☐ D `SY-INDEX`

Learning Assessment - Answers

1. Which of the following are mandatory parts of an `IF` statement?

   Choose the correct answers.

   - [X] A `IF`
   - [ ] B `ELSEIF`
   - [ ] C `ELSE`
   - [X] D `ENDIF`

   You are correct! With an `IF` construct, you can define any logical expressions as check conditions. If the `IF` condition is true, the relevant block of code is executed. Otherwise, the condition specified in the next `ELSEIF` branches is checked. If none of the specified conditions are met, the system executes the `ELSE` branch (if you have defined it). `ELSEIF` and `ELSE` branches are not mandatory, but every IF must have an `ENDIF`. For more information, see Unit 3, Lesson 1: Implementing Conditional Logic, task IF Statements.

2. Which of the following system variables contains the current loop pass in a DO loop?

   Choose the correct answer.

   - [ ] A `SY-DATUM`
   - [ ] B `SY-UNAME`
   - [ ] C `SY-UZEIT`
   - [X] D `SY-INDEX`

   You are correct! In the DO and WHILE loops, the system variable sy-index contains the number of the current loop pass. Because of this, you do not need to maintain your own iteration counter within the code. For more information, see Unit 3, Lesson 2: Implementing Loops, task Loops.

UNIT 4 Runtime Errors and Error Handling

UNIT OBJECTIVES

- Analyze runtime errors
- Implement error handling

Unit 4
Lesson 1

Analyzing Runtime Errors

LESSON OVERVIEW
This lesson shows you how to analyze a runtime error to discover problems in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Analyze runtime errors

ABAP Short Dump

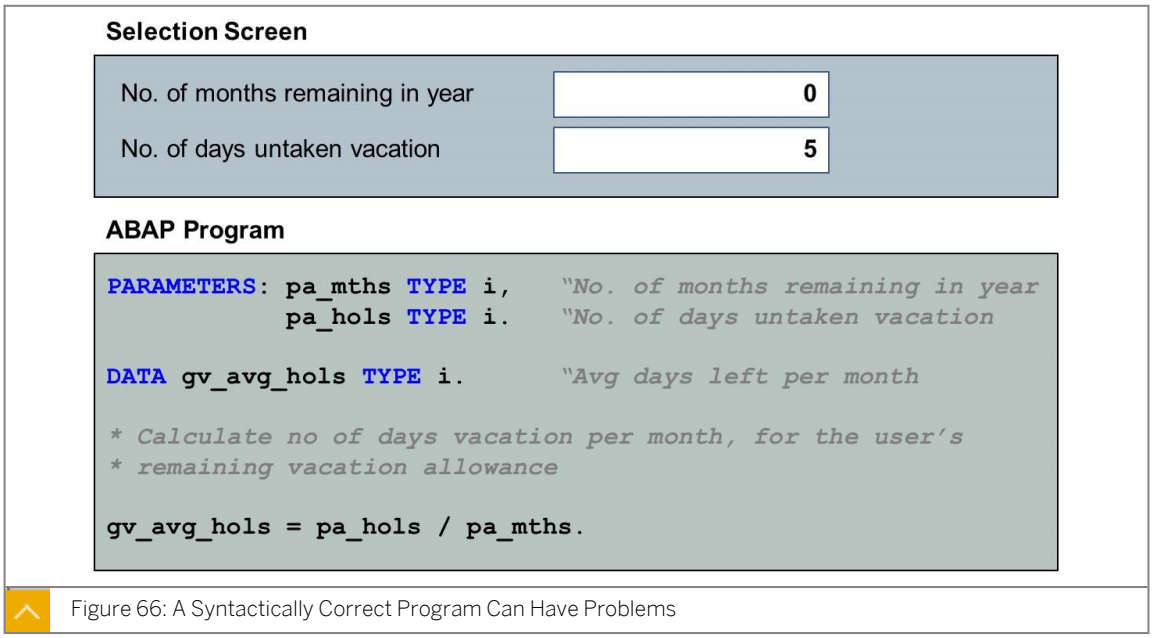

Figure 66: A Syntactically Correct Program Can Have Problems

You know that it is important to perform a syntax check regularly when writing ABAP code. However, correct syntax does not always mean your program will execute successfully at runtime.

For example, calculations in your program may cause a problem, depending on the values used. The figure, A Syntactically Correct Program Can Have Problems, demonstrates a program that, while syntactically correct, will cause a runtime error.

Runtime Error

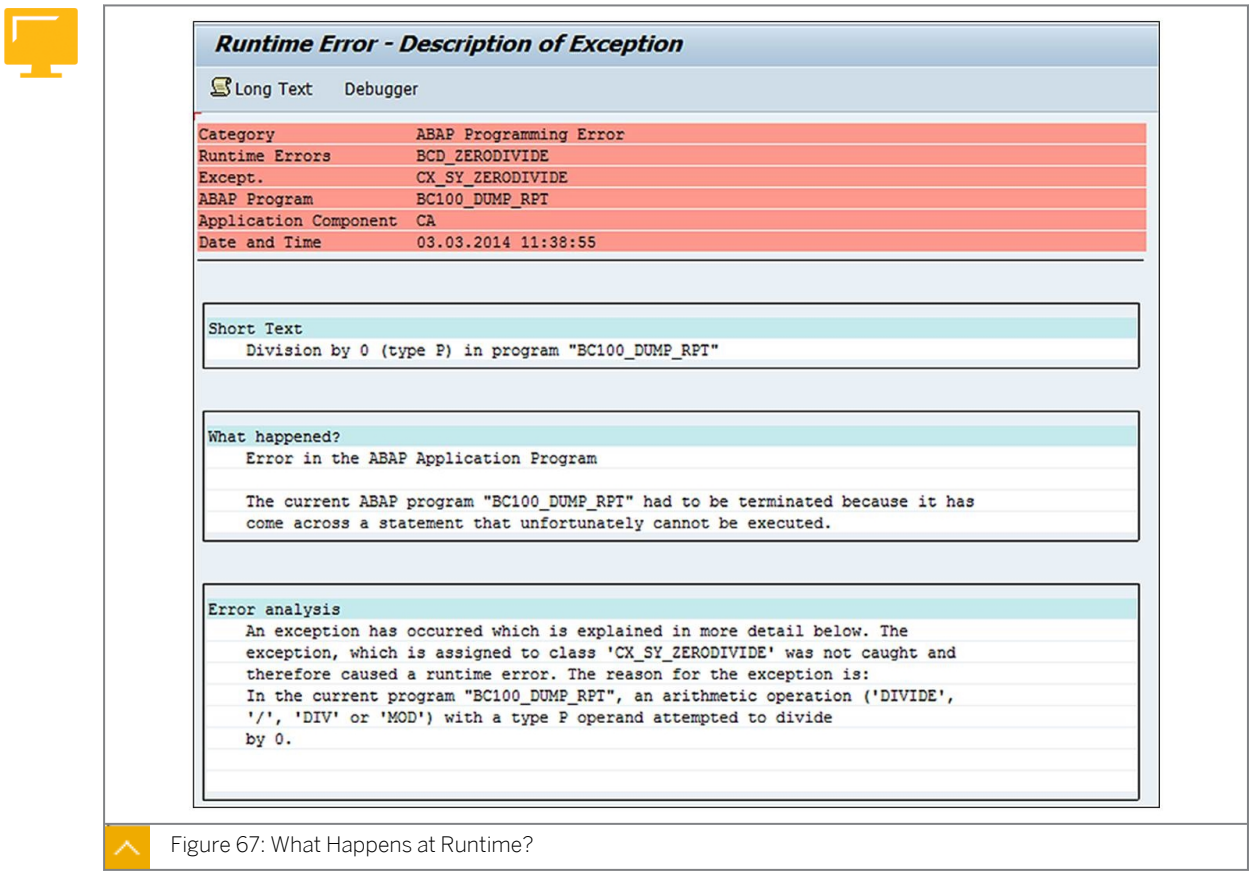

Figure 67: What Happens at Runtime?

If the runtime environment encounters a statement which cannot be executed, it terminates the program and triggers a runtime error. Each runtime error is identified by a name and assigned to a specific error situation. For example, the runtime error in the figure, What Happens at Runtime, is called BCD_ZERODIVIDE. This error relates to the fact that division by zero is impossible.

If a runtime error is not caught, the runtime environment terminates the program, and generates and displays a short dump (error log). A short dump indicates the following (among other things):

- The name of the runtime error
- A description of the problem
- The contents of any variables when the runtime error occurred
- The line of code which resulted in the termination

A short dump also allows you to navigate directly to the ABAP Debugger to see the values of variables just before the runtime error occurred.

Runtime errors are normal. You will probably encounter them frequently while developing your programs. However, it is important that you try to prevent them. A user in your production system may be confused if they see a runtime error such as the error in the figure, since the user may not understand a reference to the ABAP program and the technical information that is presented.

Short dumps are retained in the system for 14 days by default. You can view them using the transaction code `ST22`.

Source Code Extract in the Runtime Error Description

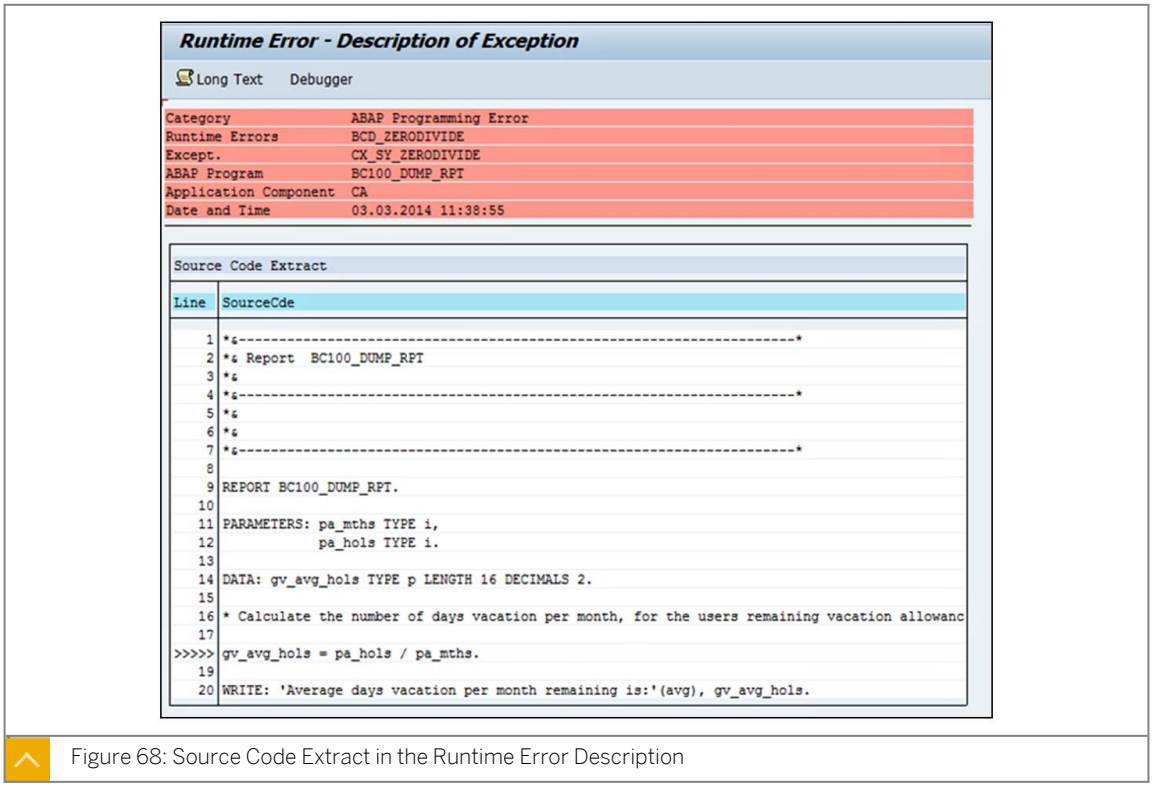

Figure 68: Source Code Extract in the Runtime Error Description

One of the most useful pieces of information in the short dump for a runtime error is the source code extract. In the extract, the line of code which could not be executed (that is, the line that triggered the runtime error) is indicated with arrows (**>>>>>>**).

Avoiding Runtime Errors

Identifying the statement which caused the issue will help you determine how to prevent the error from reoccurring.

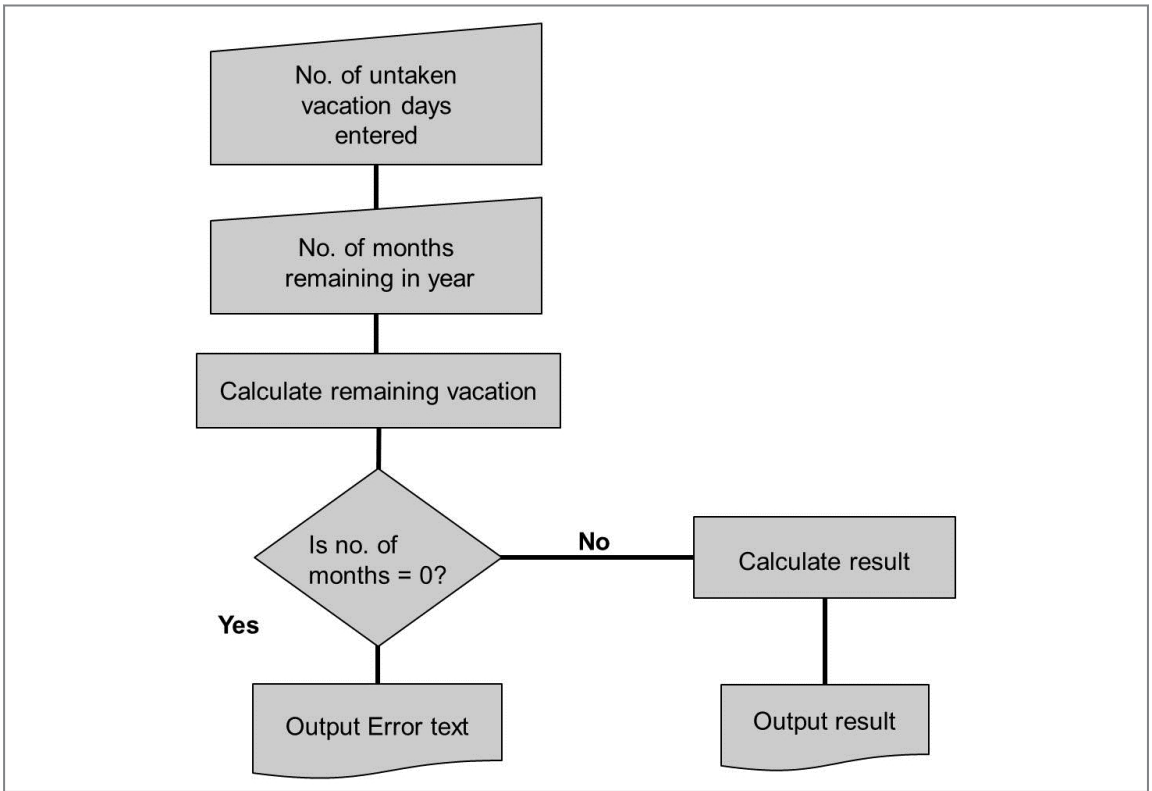

Figure 69: How to Prevent a Runtime Error

Using Conditional Logic to Avoid a Runtime Error

```
PARAMETERS: pa_mths TYPE i,   " No. of months remaining in year
            pa_hols TYPE i.   " No. of days untaken vacation

DATA gv_avg_hols TYPE i.      " Avg vacation days left per month

* Calculate no. of days vacation per month, for the user's
* remaining vacation allowance

IF pa_mths = 0.

  WRITE '0 months remaining'.

ELSE.

* Only perform calculation if pa_mths is not 0

  gv_avg_hols = pa_hols / pa_mths.

ENDIF.
```

Figure 70: Using Conditional Logic to Avoid a Runtime Error

LESSON SUMMARY

You should now be able to:

- Analyze runtime errors

Unit 4
Lesson 2

Implementing Error Handling

LESSON OVERVIEW
This lesson shows you how to implement error handling in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Implement error handling

Error Messages

So far, you have used the `WRITE` statement to output text and data in simple ABAP reports. If a problem occurs in your program, the `WRITE` statement is a simple way to output information that tells the user about the problem.

There is another way to provide information to the user. While working with SAP applications, you might see **dialog messages** that inform you about problems or information. Dialog messages can appear in a dialog box or in the status bar at the bottom of the screen.

In this lesson, we look at how you can use dialog messages in your program to provide information or error details to the user.

Format of MESSAGE Statement

The `MESSAGE` statement is formatted as follows: `MESSAGE tnnn(name_of_message_class).` In this, `t` is the message type and `nnn` is the message number.

Table 9: Message Types

Message Type	Description	Behavior	Location of Message
`i`	Information message	Program continues once user confirms the message	Dialog box
`e`	Error message	In a simple executable program, program terminates once user confirms the message	Status bar (bottom of screen)

You can issue a message with the `MESSAGE` statement. This statement is constructed as follows.

First, consider the behavior of the message. It can be an error message (for example, a situation where it makes no sense for the program to continue) or an information message (for example, information the user should know). By default, error messages display in the

status bar while information messages display in a dialog box. The letters `i` and `e` in the `MESSAGE` statement indicate the type of message.

The statement starts as follows:

- `MESSAGE i` (for an information message)
- `MESSAGE e` (for an error message)

Next, consider the message text. The system stores messages centrally in **message classes**. A message class is a collection of related messages. You can either use existing message classes or create your own. A message class can contain many message texts, and each message text is represented by a three digit number (for example, 001). To issue a particular message from your program, you must specify both the message number and the message class to which it belongs. Therefore, a complete `MESSAGE` statement looks like the following example:

```
MESSAGE i001(bc100_msgs).
```

This statement issues an information message to the user. The message text is taken from message 001 of the message class `BC100_MSGS`.

> Note:
> The demo program DEMO_MESSAGES that is delivered with the SAP standard system allows you to test the display behavior of different message types.

Message Classes

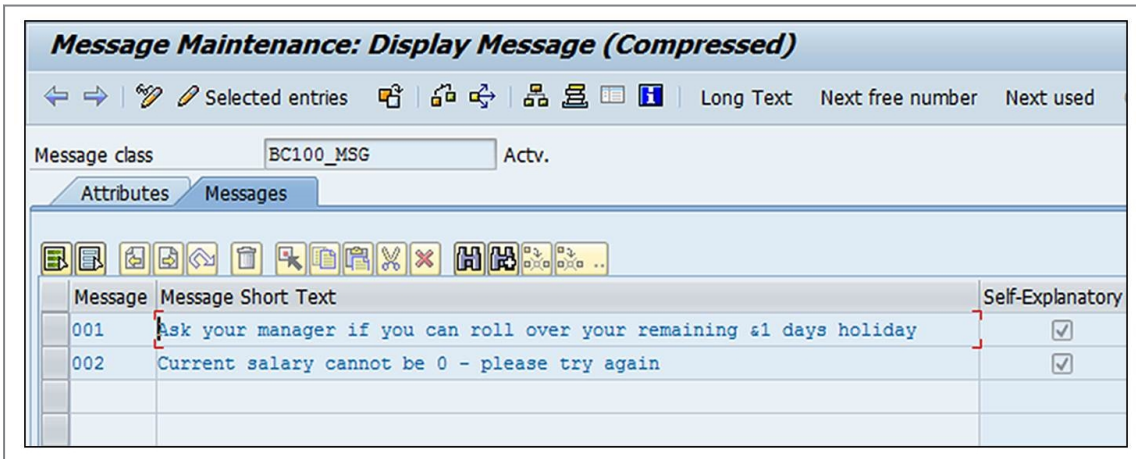

Figure 71: Message Classes

You can display or maintain a message class using the Message Maintenance tool (transaction code `SE91`). The figure Message Classes shows a message class that contains two message texts.

If you create your own message class, use the customer namespace (that is, the name of the message class must begin with **`Z`** or **`Y`**).

If you see a `MESSAGE` statement in an ABAP program, you can display the message class by double-clicking its name in the source code. You can also use this method to create a message: write the `MESSAGE` statement in the source code, then double-click the message to create it.

Message with a Placeholder

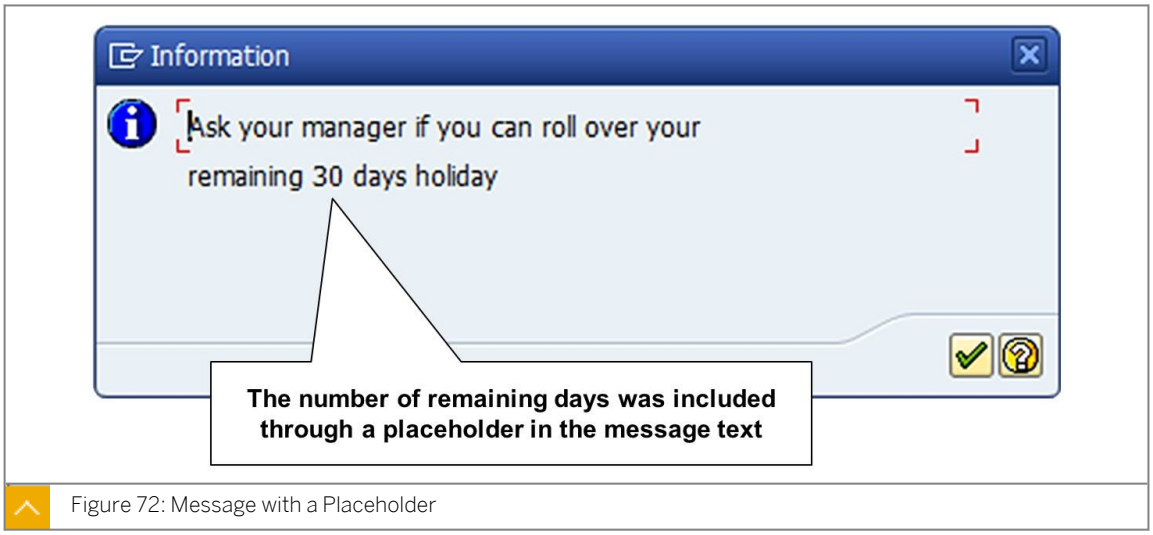

Figure 72: Message with a Placeholder

Message texts often include contextual information. The figure Message with a Placeholder shows an example: the message text includes the actual number of remaining vacation days. This is achieved by using a **placeholder** in the message text. The program must fill the placeholder with a suitable value.

A message can contain up to four placeholders. In this example, the number of vacation days replaces a placeholder that was defined in the message text.

Placeholder Definition

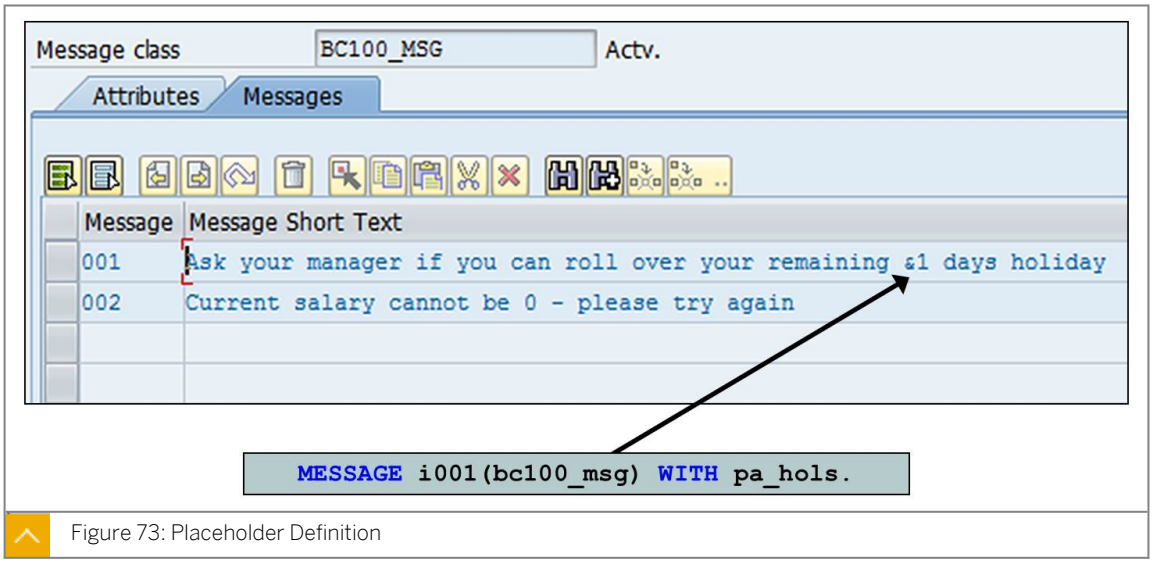

Figure 73: Placeholder Definition

In the Message Maintenance tool, placeholders are represented by an ampersand (&). If a message contains placeholders, you can fill them with values from your program by using the `WITH` addition. This addition is shown in the code in the figure Placeholder Definition. When the message displays, the value from your program is shown instead of the placeholder. In this example, a number is used with the ampersand to indicate this is placeholder number 1.

When the message in this figure displays, the `WITH` addition replaces the placeholder `&1` with the value of `pa_hols`.

LESSON SUMMARY

You should now be able to:

- Implement error handling

Learning Assessment

1. A runtime error means that all statements in the ABAP program were successfully executed.

   Determine whether this statement is true or false.

   - [ ] True
   - [ ] False

2. When you write a `MESSAGE` statement, you must supply several pieces of information. Put these pieces of the statement in the correct sequence.

   Arrange these steps into the correct sequence.

   - [ ] Message number
   - [ ] Message class
   - [ ] `MESSAGE` keyword
   - [ ] Message type

Learning Assessment - Answers

1. A runtime error means that all statements in the ABAP program were successfully executed.

   Determine whether this statement is true or false.

   ☐ True

   ☒ False

   You are correct! If the runtime environment encounters a statement which cannot be executed, it terminates the program and triggers a runtime error. The statements following the statement that caused the error are not executed. For more information, see Unit 4, Lesson 1: Analyzing Runtime Errors, task Runtime Error.

2. When you write a `MESSAGE` statement, you must supply several pieces of information. Put these pieces of the statement in the correct sequence.

   Arrange these steps into the correct sequence.

   [3] Message number

   [4] Message class

   [1] `MESSAGE` keyword

   [2] Message type

   You are correct! The `MESSAGE` statement is formatted as follows: keyword MESSAGE, followed by tnnn (name_of_message_class). In this, t is the message type and nnn is the message number. For more information, see Unit 4, Lesson 2: Implementing Error Handling, task Format of MESSAGE Statement.

UNIT 5 Additional ABAP Programming Techniques

UNIT OBJECTIVES

- Retrieve data from the database
- Describe modularization in ABAP
- Call a function module from a program

Retrieving Data From the Database

LESSON OVERVIEW
In this lesson, you learn how to retrieve data from the database and use it in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Retrieve data from the database

Simple Data Retrieval

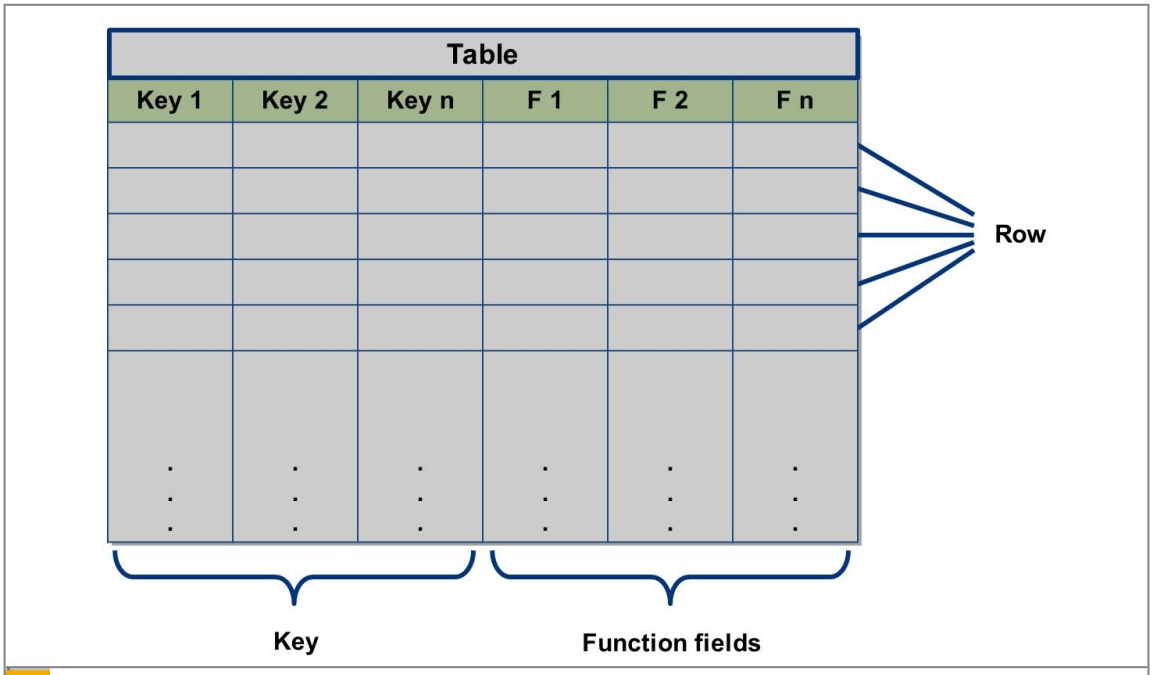

Figure 74: Database Tables

Most ABAP programs that you develop will need to work with data from the SAP database (for example, master data or transactional data). Data retrieval is a complex topic. For example, database accesses can have a huge impact on the performance of an ABAP program.

The topic of data retrieval is not covered in depth in this course. Instead, this course provides a general idea of the syntax you can use and the things that you need to consider. This lesson looks at simple data retrieval of single records from the database.

When retrieving data, the first thing you must do is identify the database table that stores the data. The figure Database Tables illustrates the structure of a database table.

The ABAP Dictionary

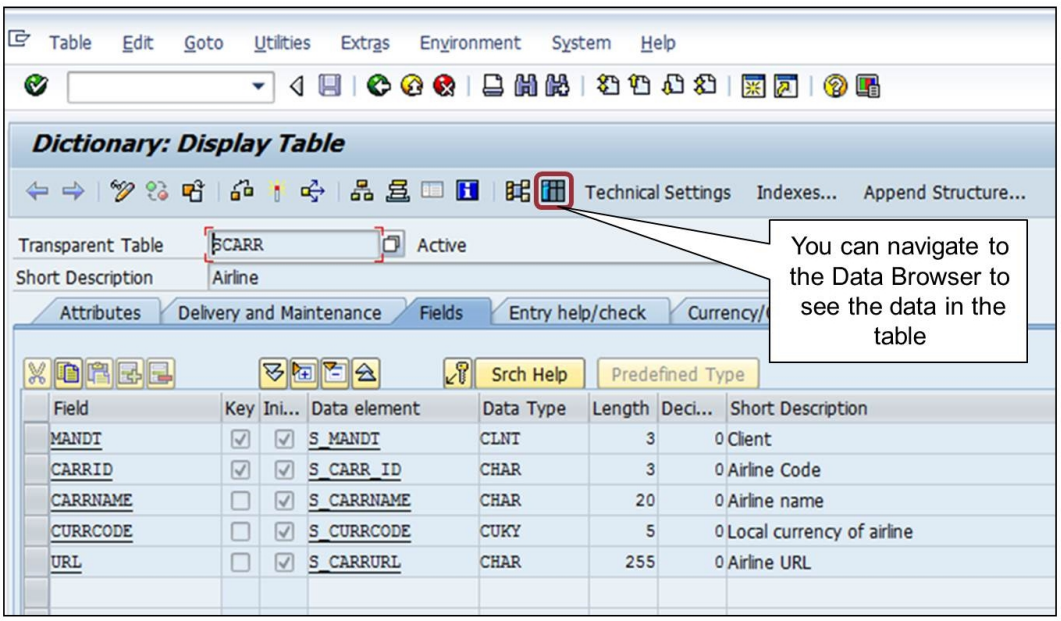

Figure 75: The ABAP Dictionary

The ABAP Dictionary (transaction code `SE11`) provides a central point for the management of all database tables in your SAP system. It also provides a number of other important functions.

When you want to retrieve data from the database, you must know some basic information about the table that contains the data. The ABAP Dictionary displays the following:

- The fields (or columns) that make up the database tables
- Details of their technical characteristics (that is types, lengths, and the fields that make up the **key** of the database table)

To view the **contents** of the table, you can branch to the Data Browser (transaction code `SE16`). This tool is useful for verifying data when testing your ABAP programs.

Data Retrieval in ABAP

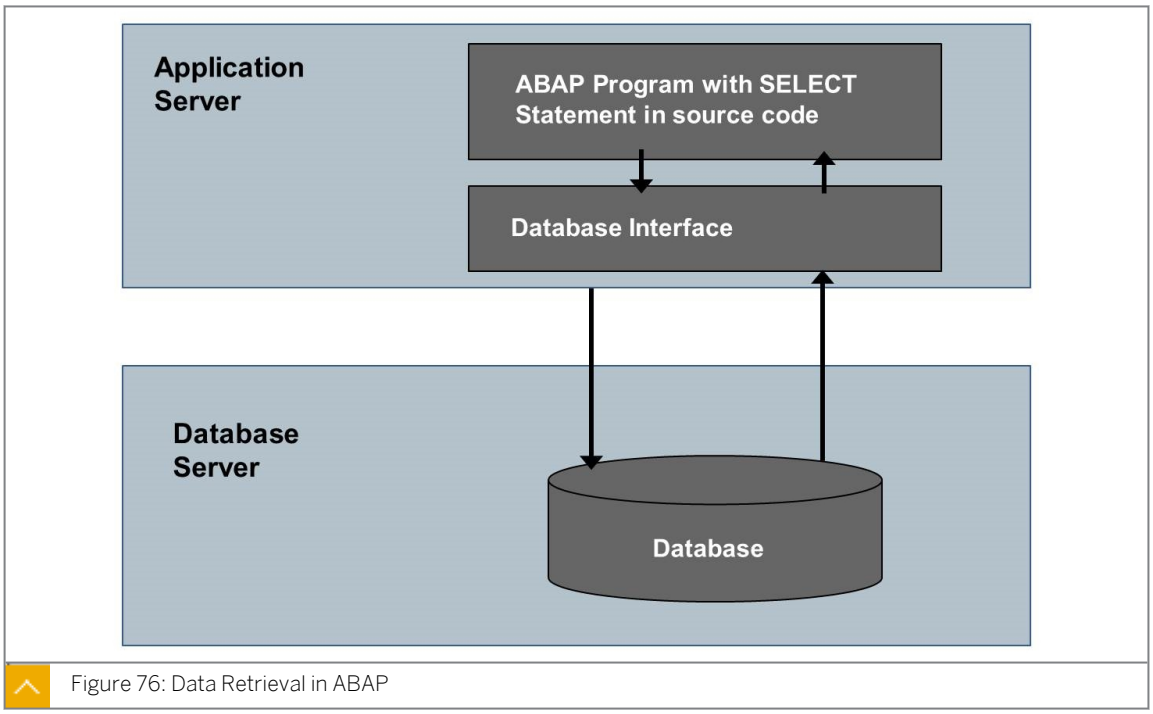

Figure 76: Data Retrieval in ABAP

To read data from the database in your program, code a `SELECT` statement. When your program runs, the Database Interface translates the `SELECT` statement into native SQL, then transfers it to the database. The Database Interface also feeds the retrieved data back to your program. The interaction is illustrated in the figure Data Retrieval in ABAP.

The SELECT Statement

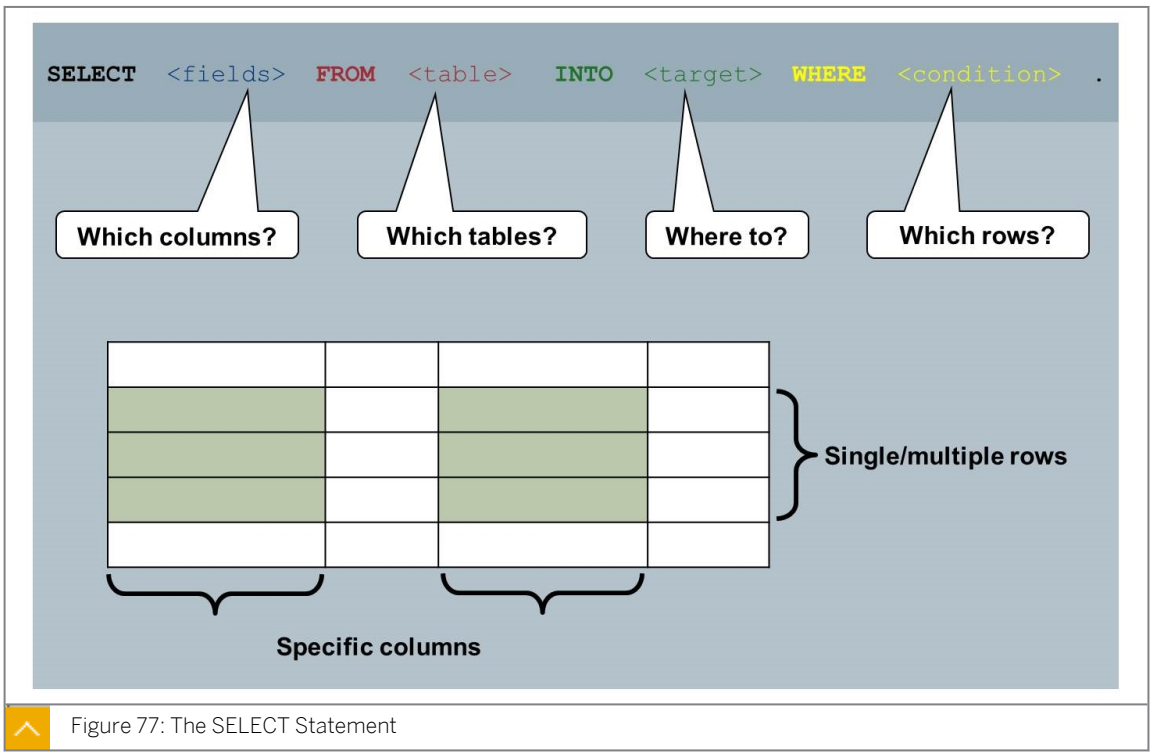

Figure 77: The SELECT Statement

There are many variations of the SELECT statement, but they all have the same basic structure. This structure is shown in the figure The SELECT Statement.

Each clause of the statement has a different task:

- SELECT
- FROM

  Names the source (database table or view) from which the data is to be selected.
- INTO

  Determines the target variables, structure, or internal table into which the selected data is to be placed.
- WHERE

  Specifies the columns of the table that are to be selected.

When you use a SELECT statement, you must define where the Database Interface will place the data. For this, you need a suitable variable, structure, or internal table. The type of storage depends on the type and quantity of data you retrieve and the variation of the SELECT statement that you used.

This course focuses on the SELECT SINGLE variation of the SELECT statement.

The SELECT SINGLE Statement

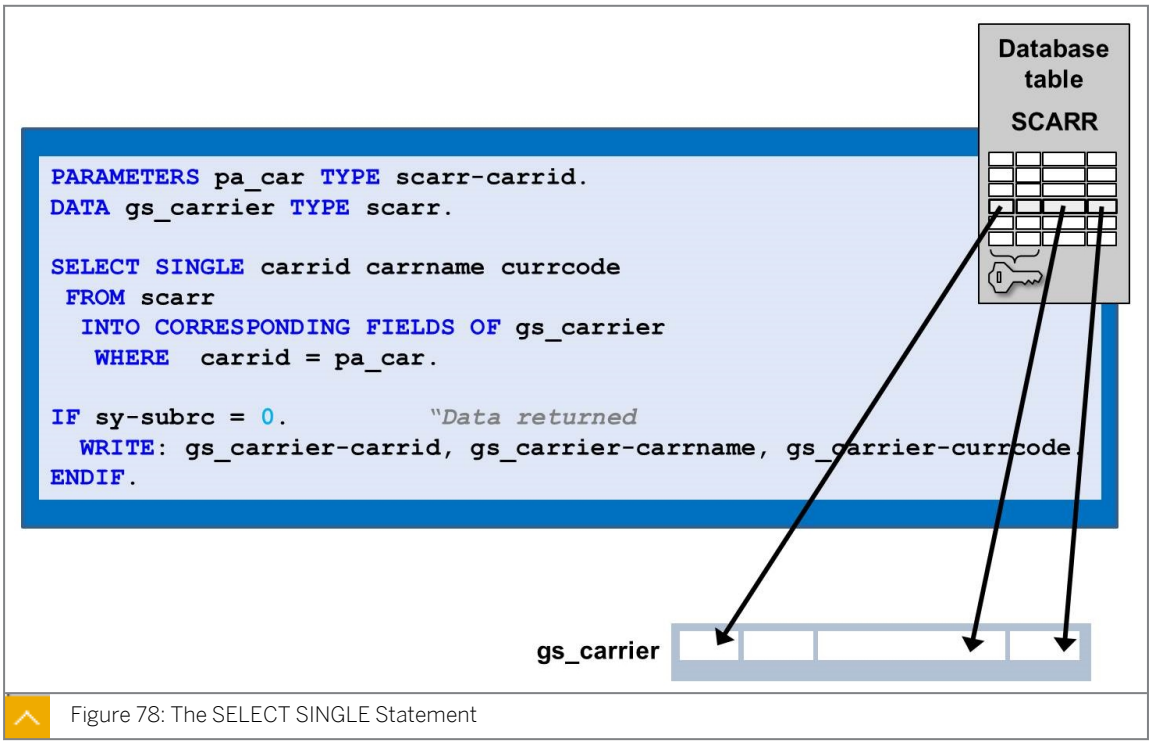

Figure 78: The SELECT SINGLE Statement

You can use the SELECT SINGLE statement to read a single record from the database table.

With a SELECT SINGLE, the full business key of the table is often provided in the WHERE clause. This ensures that the single unique record that is required is fetched from the database.

In the figure The `SELECT SINGLE` Statement, the developer defined a structure (`gs_carrier`) that has the same components as the database table (`scarr`).

The `SELECT SINGLE` statement reads three fields from that database table: `carrid`, `carrname`, and `currcode`.

The `INTO` clause specifies the structure in which the returned data will be stored. `INTO CORRESPONDING FIELDS OF` tells the database interface to put the data into the fields of the structure (`gs_carrier`) that have the same names as the fields retrieved from the database table (`scarr`).

If the system finds a suitable record, the system variable `sy-subrc` returns the value **0**. If this occurs, the program outputs the data that was retrieved.

LESSON SUMMARY

You should now be able to:

- Retrieve data from the database

Unit 5
Lesson 2

Describing Modularization in ABAP

LESSON OVERVIEW
This lesson introduces modularization and shows you how it can simplify the development process.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Describe modularization in ABAP

Modularization

Although a good programmer can, given time to do so, create an entire program for almost any realistic request, you do not have to code everything in ABAP yourself. A smart programmer assumes that at least parts of the necessary functionality have already been developed.

Developing an entire program from scratch is time-consuming and difficult. Use modularization techniques (where possible) to reuse certain code blocks and speed up development time.

A modularization unit is a source code block that encapsulates a particular function. With modularization units, you can use a function many times in many locations without implementing the full source code each time.

Modularization makes it easier for you to maintain programs, since any required changes are made once (in the relevant modularization unit) and not at multiple points in one or many ABAP programs.

The Principle of Modularization (1)

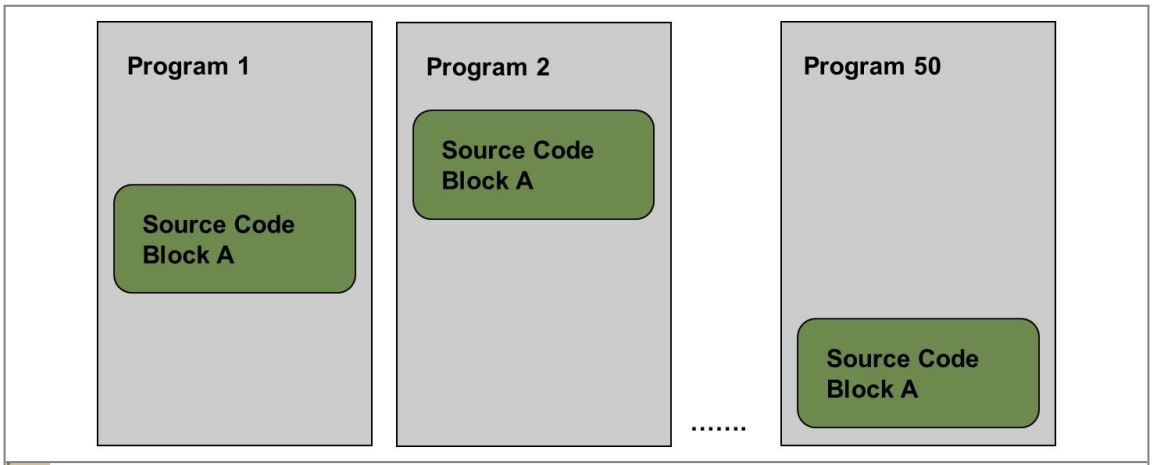

Figure 79: The Principle of Modularization (1)

The figure The Principle of Modularization (1) shows an example where the same block of source code is used in multiple programs. This block consists of 100 lines of code. If you need to change that code, you must find the same block and make the change in every program that uses it.

The Principle of Modularization (2)

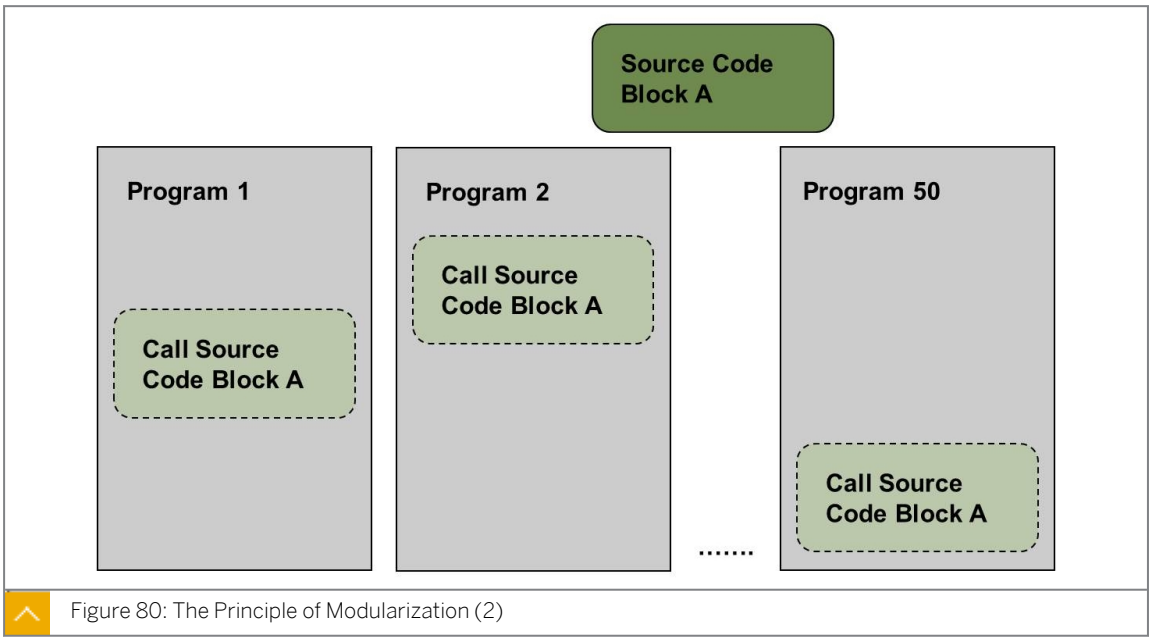

Figure 80: The Principle of Modularization (2)

The figure The Principle of Modularization (2) shows how modularization can reduce the work required to maintain that code. The 100-line block of code is maintained once in a central location. Other programs can then call that block of code when and where they need it.

Modularization Data Exchange (1)

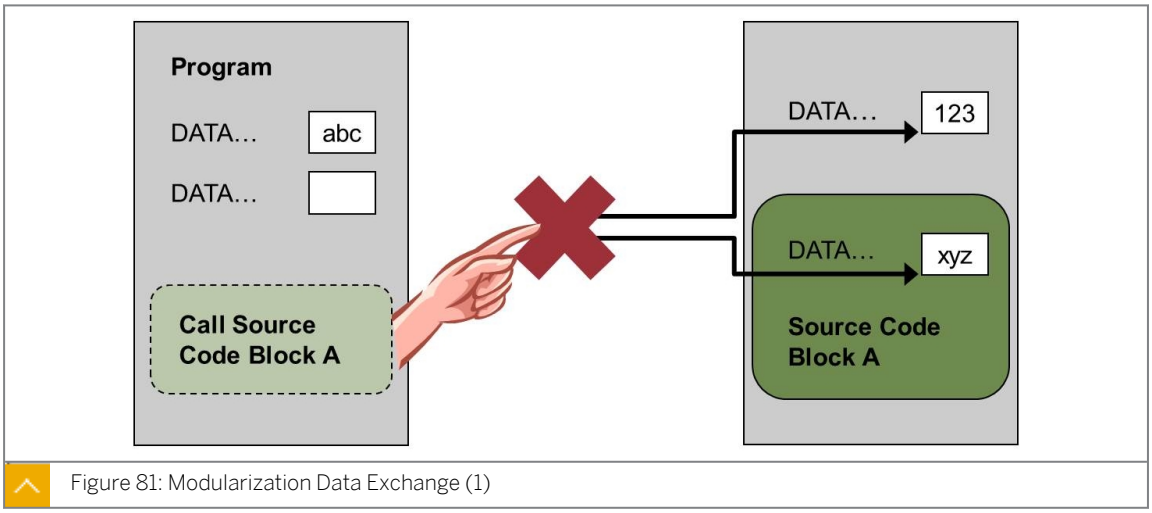

Figure 81: Modularization Data Exchange (1)

A modularization unit represents a specific function. To perform this function, the unit typically requires the calling program to send data, and the calling program expects the modularization unit to return data. Neither the calling program nor the modularization unit can access the data of the other directly, as shown in the figure Modularization Data Exchange (1).

Modularization Data Exchange (2)

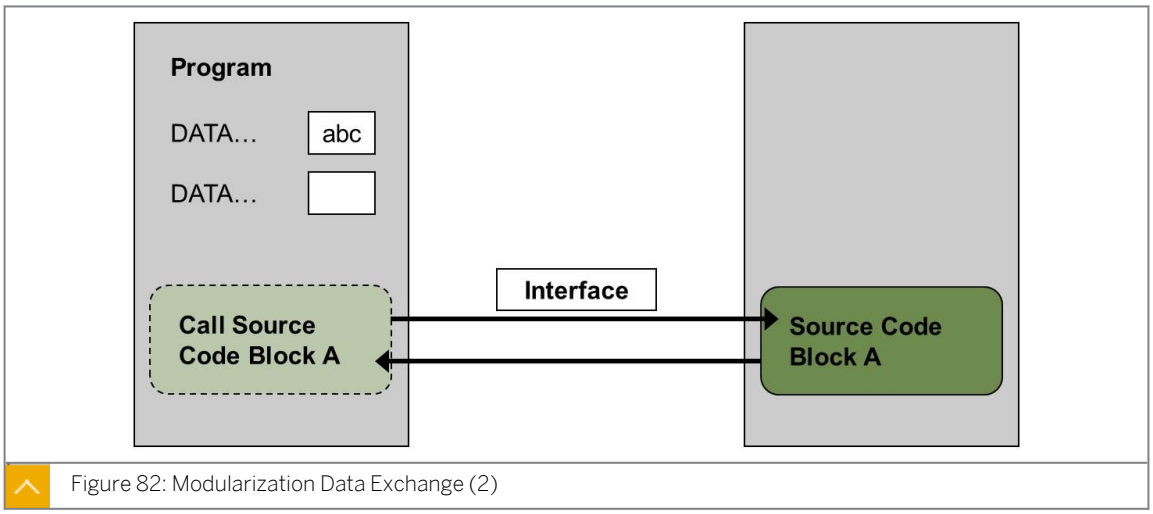

Figure 82: Modularization Data Exchange (2)

Instead, the modularization unit has an **interface**. The interface describes the information required by the modularization unit (that is, the data that the calling program must send) and the results that will be returned.

The interface consists of parameters that are used to exchange data between the program and the modularization unit. Parameters are distinguished by whether they pass data to the modularization unit or return data to the calling program.

LESSON SUMMARY

You should now be able to:

- Describe modularization in ABAP

Unit 5
Lesson 3

Using Function Modules

LESSON OVERVIEW
In this lesson, you learn how to evaluate a function module and call it in a program.

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Call a function module from a program

Function Modules

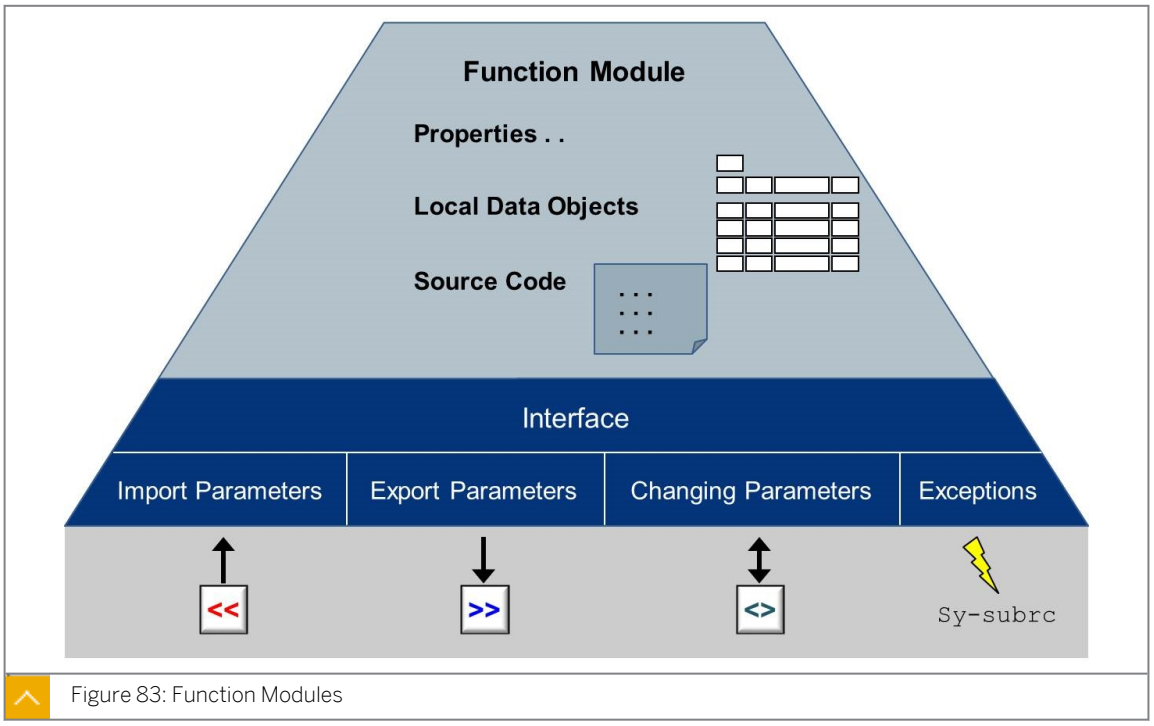

Figure 83: Function Modules

Function modules are one of the most commonly used type of modularization unit in SAP. You can use the Function Builder tool (transaction code `SE37`) to create and analyze function modules.

Consider the following example: in your program, you want to calculate the number of vacation days remaining for a user. A search in the system reveals that a function module **BC100_CALC_HOLS** might be useful for this purpose.

Note:
This course does not cover how to search for function modules in the system. For more information, see the course ABAP Workbench Foundations (BC400).

The Function Builder

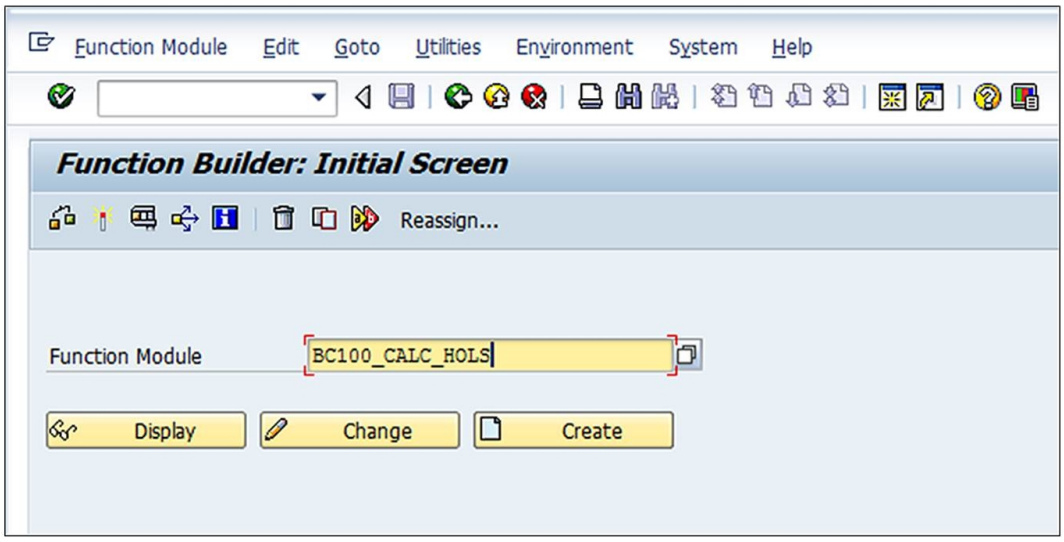

Figure 84: The Function Builder

To analyze whether the function module BC100_CALC_HOLS suits your requirements, start the Function Builder and enter the function module name.

The *Attributes* tab page provides some useful information (including a short description) to help you determine if the function module is suitable.

Analyzing the Data Required from the Calling Program

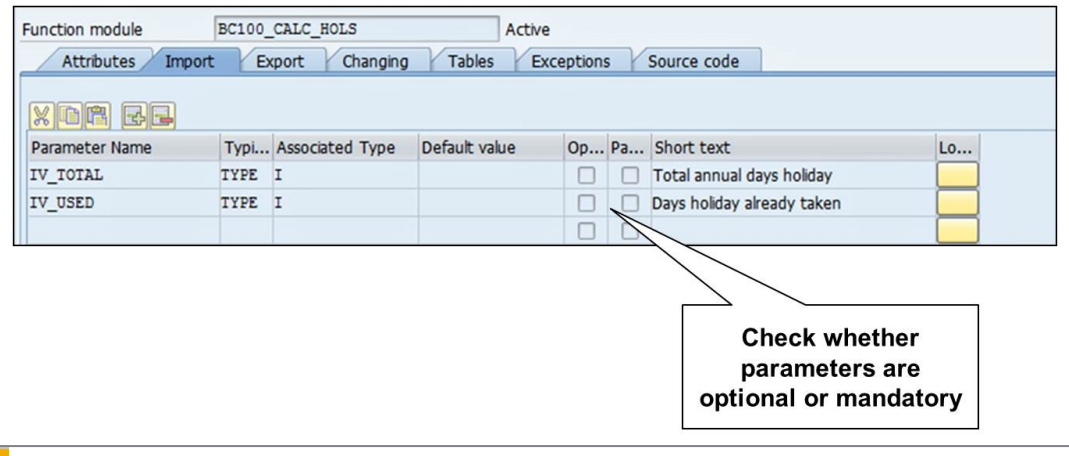

Figure 85: Analyzing the Data Required from Calling Program

Next, examine the function module interface. If your program cannot provide the required information, or if the function module does not return the information you need, it is not suitable.

A function module interface can consist of *Import*, *Export*, and *Changing* parameters. An interface can also include *Exceptions*.

Import parameters specify the data that the function module requires from the calling program. Export parameters specify the data that is returned to the calling program.

Mandatory parameters must be supplied with data when the function module is called. For more information about the values you should supply for the parameters, see the function module documentation and documentation for the individual parameters.

The import and export parameters for BC100_CALC_HOLS are shown in the figures Analyzing the Data Required from Calling Program and Analyzing the Data Returned by the Function Module.

Analyzing the Data Returned by the Function Module

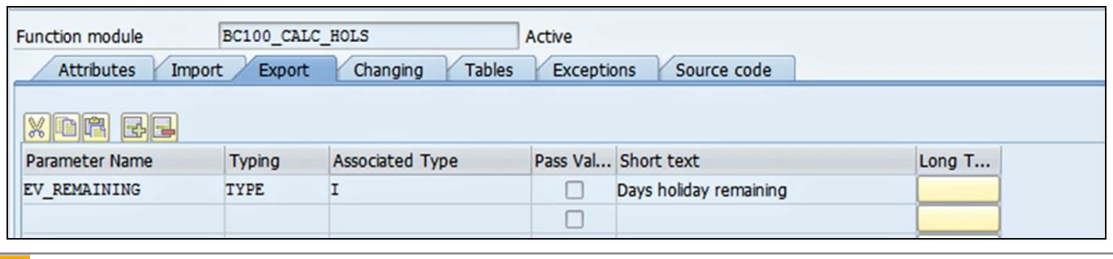

Figure 86: Analyzing the Data Returned by the Function Module

Testing Function Modules

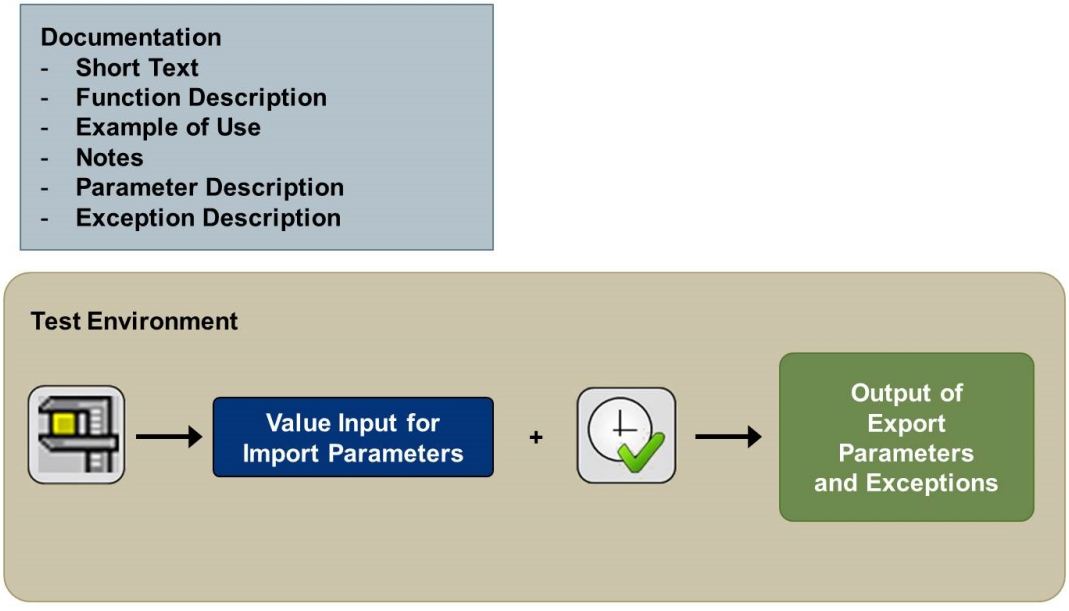

Figure 87: Testing Function Modules

You can also evaluate a function module by testing it in the Function Builder. The test environment allows you to enter test values for any import parameters, execute the function module, and view the data returned through the export parameters. If the function module triggers an exception due to an error, the exception displays. To test a function module, choose *Test*.

You can also access the function module documentation (if it exists in your logon language) by choosing the *Function Module Documentation* button.

In display mode, you can open the parameter documentation by clicking the displayed links. You can also access the parameter documentation by choosing the relevant button in the *Long Text* column. If a green light icon is visible beside a parameter, documentation exists for that parameter.

By reading the documentation and testing the function module, you can understand how the function module works and decide if it is suitable for your requirements.

Call a Function Module

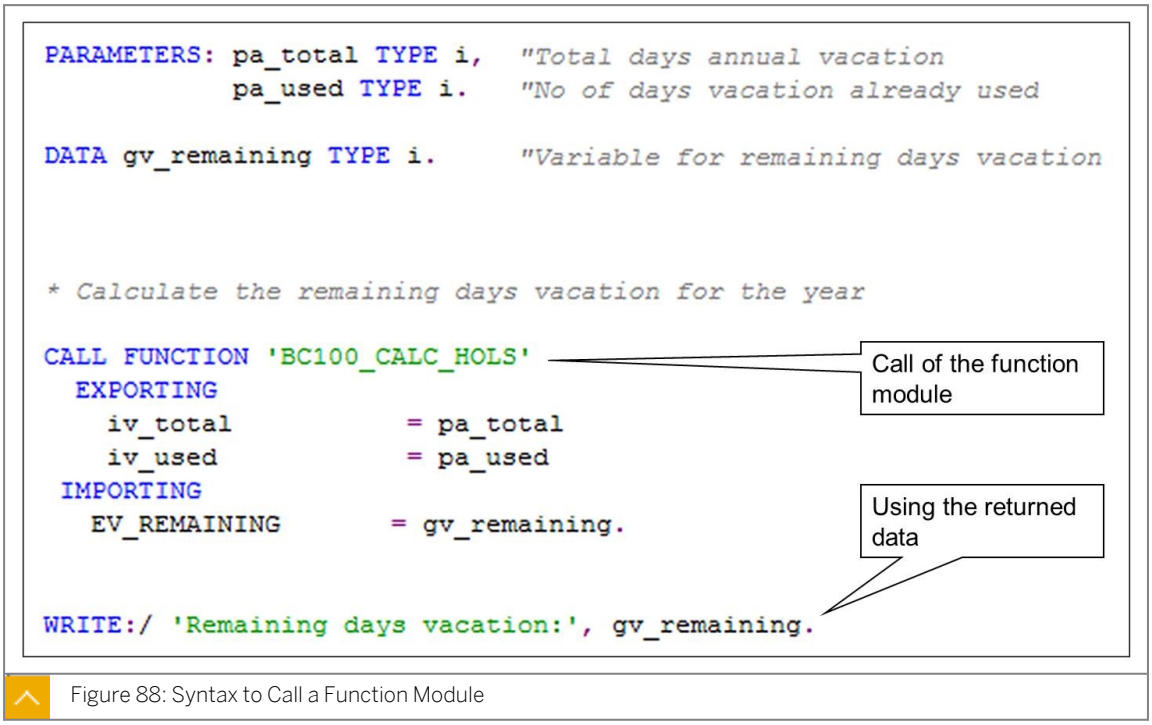

Figure 88: Syntax to Call a Function Module

You can call a function module in your program with the `CALL FUNCTION` statement. Follow the keyword with the name of the function module in capital letters, enclosed by single quotes.

In the `EXPORTING` block, the calling program must pass suitable values to the import parameters of the function module. The calling program exports the data to the function module (hence the name of the block), and the function module imports the data.

In the `IMPORTING` block, actual variables are assigned to the export parameters. The function module exports data to the calling program, and the calling program imports the data (hence the name of the block). After the function call, you can access the contents of these variables.

In the call syntax, specify the name of the function module parameter on the left and the value or variable supplied by the calling program on the right.

The figure Syntax to Call a Function Module shows a complete call to a function module. The program exports the values of `pa_total` and `pa_used` to the function module, which performs a calculation using the values. The result of the calculation is returned to the program and stored in the variable `gv_remaining`. This value is then output using the `WRITE` statement.

Simplify Calling Function Modules

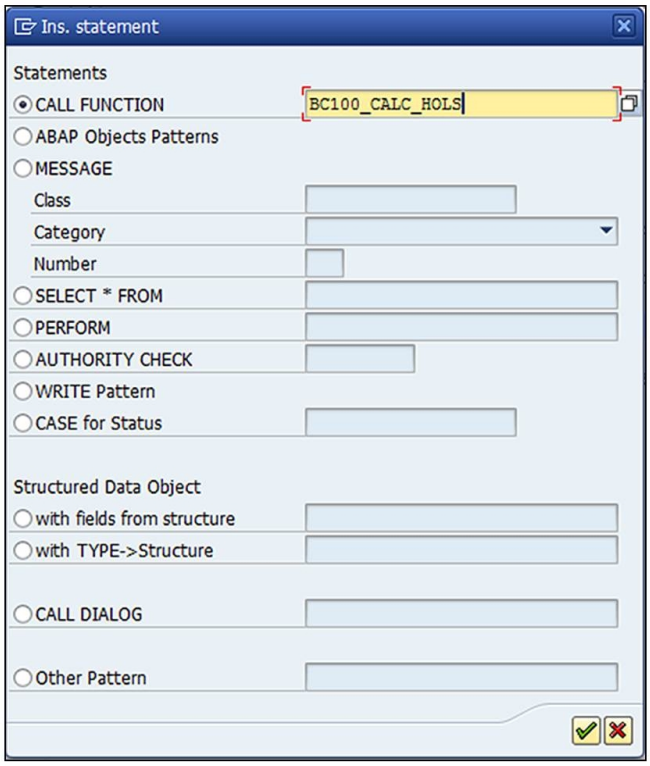

Figure 89: Simplify Calling Function Modules

The *Pattern* button on the ABAP Editor toolbar simplifies the task of inserting a `CALL FUNCTION` statement into your code. Choose this button and enter the name of the function module that you want to call. The system inserts the skeleton code into your program. Supply appropriate values or variables for the parameters, and the function call is complete.

The figure Simplify Calling Function Modules shows the dialog box that displays when you choose *Pattern*.

LESSON SUMMARY

You should now be able to:

- Call a function module from a program

Learning Assessment

1. Which system variable is set to 0 if a `SELECT` is successful and data is retrieved from the database?

   Choose the correct answer.

   - ☐ A `SY-SUBRC`
   - ☐ B `SY-INDEX`
   - ☐ C `SY-DATUM`
   - ☐ D `SY-UNAME`

2. Which of the following are used to exchange data between a calling program and a modularization unit?

   Choose the correct answer.

   - ☐ A System variables
   - ☐ B Parameters
   - ☐ C `SELECT` statements
   - ☐ D Literals

3. When calling a function module, the `EXPORTING` block in your source code passes data to the export parameters of the function module.

   Determine whether this statement is true or false.

   - ☐ True
   - ☐ False

Learning Assessment - Answers

1. Which system variable is set to 0 if a `SELECT` is successful and data is retrieved from the database?

   Choose the correct answer.

   - [X] A `SY-SUBRC`
   - [ ] B `SY-INDEX`
   - [ ] C `SY-DATUM`
   - [ ] D `SY-UNAME`

   You are correct! If the system finds a suitable record, the system variable `sy-subrc` returns the value 0. If this occurs, the program outputs the data that was retrieved. For more information, see Unit 5, Lesson 1: Retrieving Data from the Database, task The SELECT SINGLE Statement.

2. Which of the following are used to exchange data between a calling program and a modularization unit?

   Choose the correct answer.

   - [ ] A System variables
   - [X] B Parameters
   - [ ] C `SELECT` statements
   - [ ] D Literals

   You are correct! The interface of a modularization unit consists of parameters that are used to exchange data between the program and the modularization unit. Parameters are distinguished by whether they pass data to the modularization unit or return data to the calling program. For more information, see Unit 5, lesson 2: Describing Modularization in ABAP, task Modularization Data Exchange.

3. When calling a function module, the `EXPORTING` block in your source code passes data to the export parameters of the function module.

   Determine whether this statement is true or false.

   ☐ True

   ☒ False

   You are correct! In the `EXPORTING` block, the calling program must pass suitable values to the import parameters of the function module. The calling program exports the data to the function module (hence the name of the block), and the function module imports the data. For more information, see Unit 5, lesson 2: Describing Modularization in ABAP, task Call a Function Module.