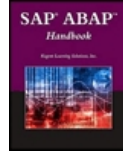


Chapters *To Go*



SAP ABAP Handbook

by Kogent Learning Solutions, Inc.
Jones and Bartlett Publishers. (c) 2010. Copying Prohibited.

Reprinted for Julio De Abreu Molina, IBM

jdeabreu@ve.ibm.com

Reprinted with permission as a subscription benefit of **Books24x7**,
<http://www.books24x7.com/>

All rights reserved. Reproduction and/or distribution in whole or in part in electronic, paper or other forms without written permission is prohibited.



Chapter 6: ABAP Programming in ABAP Editor

Overview

ABAP is a fourth-generation programming language used to develop ABAP programs, such as user dialogs and reports. The ABAP language provides a collection of ABAP statements, such as `DATA` and `TABLES`, to create programs. ABAP programs are created to process data within the dialog steps of an application. An ABAP program is not a single sequential unit of ABAP statements but a collection of various processing blocks that may occur in any order in the source code of the program. A processing block represents a single module of a program. It can be called either from outside an ABAP program or from another processing block in the same ABAP program.

ABAP programs are created by using the `ABAP Editor` tool of `ABAP Workbench`. Through `ABAP Editor`, you can create various types of programs, such as executable programs, module pools, and subroutine pools. The source code of an ABAP program can contain declaring and assigning statements, such as `MOVE` and `WRITE TO`, and commented text. Moreover, ABAP programs can be created using various formatting statements, such as `FORMAT` and `COLORS`, so that you can get the output of a program in a user-defined format.

In this chapter, you learn about the structure of an ABAP program. You also learn about the `ABAP Editor` tool, which is used to create, display, and modify ABAP programs. Next, you learn to add comments in an ABAP program. This chapter also describes data types and data objects, and explores how to declare variables in programs. In addition, the chapter describes various groups of types in the ABAP language. You also learn about the various kinds of assignment statements, such as `MOVE`, `MOVE-CORRESPONDING`, `WRITE TO`, and `CLEAR`. Next, you learn about the different formatting options, such as the `FORMAT`, `WRITE`, and `COLORS` statements, available in ABAP. Finally, you learn about the flow of control statements, such as `IF`, `CASE`, and loop construction.

Structure of an ABAP Program

The structure of an ABAP program includes the following:

- Introductory program part
- Global declaration part
- Processing blocks (consisting of different functions, such as procedures, dialog box, and event blocks)

Figure 6.1 shows the structure of an ABAP program:

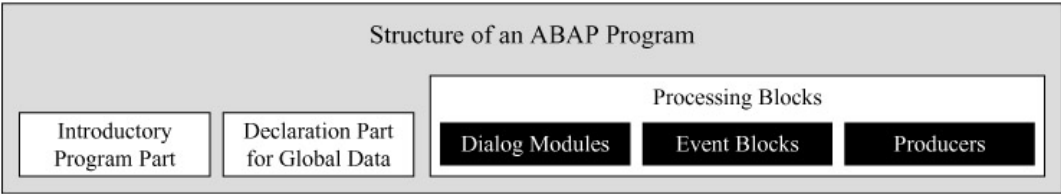


Figure 6.1: Structure of an ABAP program

The processing blocks in the ABAP program structure consist of dialog modules, event blocks, and procedures. Now, let's discuss each part of the structure of an ABAP program in detail.

Introductory Program Part

Every ABAP program begins with an introductory statement, such as `REPORT`, `PROGRAM`, or `FUNCTION-POOL`. Each program has a type, such as executable program, module pool program, or function group program, associated to it. The introductory statement of an ABAP program depends on the type of the ABAP program. Table 6.1 shows the introductory statements for different types of programs:

Table 6.1: Introductory statement

Introductory Statement	Description of the Program Type
------------------------	---------------------------------

REPORT	Represents an executable program.
PROGRAM	Represents a module pool program and subroutines.
FUNCTION-POOL	Represents a function group.
CLASS-POOL	Represents a class pool program.
INTERFACE-POOL	Represents an interface pool program.
TYPE-POOL	Defines the type group.

When a program is created, the SAP system automatically generates the most appropriate introductory statement for that program. Since the introductory statement depends on the type of the program created, the introductory statement must be assigned manually to the type of the program defined in the program properties.

Declaration Part for Global Data, Class Definitions, and Selection Screens

You always declare global data, classes, and selection screens after the introductory program part of an ABAP program. The declaration of these objects includes:

- **Global data**— Defines global data in an ABAP program. Global data is specified by declaration statements, such as `TYPES`, `TABLES`, and `DATA`. The global data defined in a program is visible in all the internal processing blocks of the program.
- **Selection screens**— Represent special screens that are generated by using ABAP statements instead of Screen Painter. These screens allow you to enter either a single value for one or more fields or a selection criterion.
- **Local class definitions**— Contains the definitions of local classes in an ABAP program. Local class definitions are created with the help of the `CLASS DEFINITION` statement. The local classes are a part of the ABAP Objects, which are object-oriented extensions of ABAP.

Note You learn more about ABAP Objects in Appendix A.

Processing Blocks

Another part of an ABAP program structure is the processing block. A processing block is a set of ABAP statements that represents a module of an ABAP program. As we know, ABAP programs are created to process data within the dialog steps of an application. This means an ABAP program is divided into numerous separate sections, which are interlinked in the respect of an application and are assigned to the respective dialog steps. In other words, we can say that ABAP programs have modular structure, where each module is represented by a processing block.

The types of processing blocks used in an ABAP program are as follows:

- Dialog modules
- Event blocks
- Procedures, including methods, subroutines, and function modules

Dialog modules and procedures are enclosed within ABAP statements such as `MODULE . . . ENDMODULE` and `FUNCTION . . . ENDFUNCTION`.

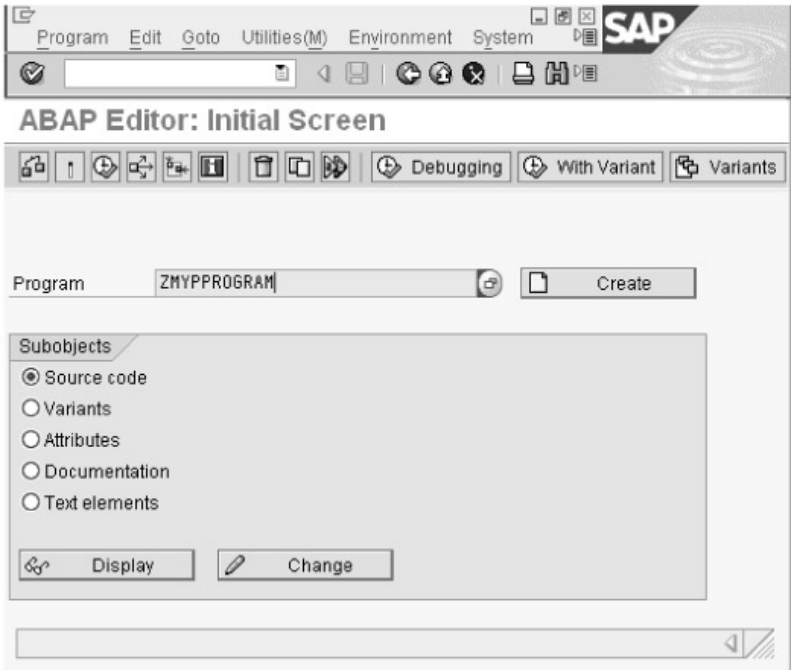
Event blocks are processed when events are triggered either by performing user actions on selection screens and lists or by the running environment of ABAP. A event block is introduced within ABAP statements, such as `START-OF-SELECTION` and `ATUSER-COMMAND`. It is created within a processing block, which starts with an ABAP statement, such as `WRITE` and `NEW-PAGE`. Note that an event block is terminated implicitly with the beginning of the next processing block, which is introduced by using another ABAP statement. In an ABAP program, all ABAP statements, except declarative statements, are a part of a processing block. In addition, ABAP statements placed between the declaration of global data and a processing block are assigned automatically to the `START-OF-SELECTION` processing block.

Because dialog modules and event blocks can be used outside an ABAP program, a processing block can be called from outside the associated ABAP program. In addition, the processing block can be called by using an ABAP command, such as `CALL METHOD`, `CALL TRANSACTION`, `SUBMIT`, or `LEAVE TO`. Note that procedures are called by using ABAP statements in an ABAP program.

Note Declarative statements are used to define the data types or data objects in an ABAP program. Some examples of declarative statements are `DATA`, `TABLES`, and `TYPES`. You learn more about these statements later in the chapter.

ABAP Editor

ABAP Editor is used to create, display, and modify ABAP programs. The initial screen of the ABAP Editor can be displayed either from the `SAP Easy Access` screen or by entering the `SE38` transaction code in the Command field (present on the standard toolbar) and pressing the `ENTER` key. [Figure 6.2](#) shows the initial screen of ABAP Editor:



© SAP AG. All rights reserved.

Figure 6.2: Initial screen of ABAP editor

In the `Program` text field of the initial screen, you can enter the name of the program to be created, displayed, or changed. [Figure 6.2](#) shows `ZMYPPROGRAM` as the name of a program in the `Program` field.

In ABAP programs, the naming conventions are as follows:

- The length of the name of an ABAP program can be from 1 to 30 characters.
- Symbols (`.`, `()`, `'`, `"`, `*`, `%`, `-`) and accented characters or German umlauts (`à`, `é`, `ø`, `ä`, `ß`, and so on) cannot be used while naming an ABAP program.

The `Subobjects` group box of the initial screen of ABAP Editor contains the `Source code`, `Variants`, `Attributes`, `Documentation`, and `Text elements` radio buttons. [Table 6.2](#) describes these radio buttons of the `Subobjects` group box:

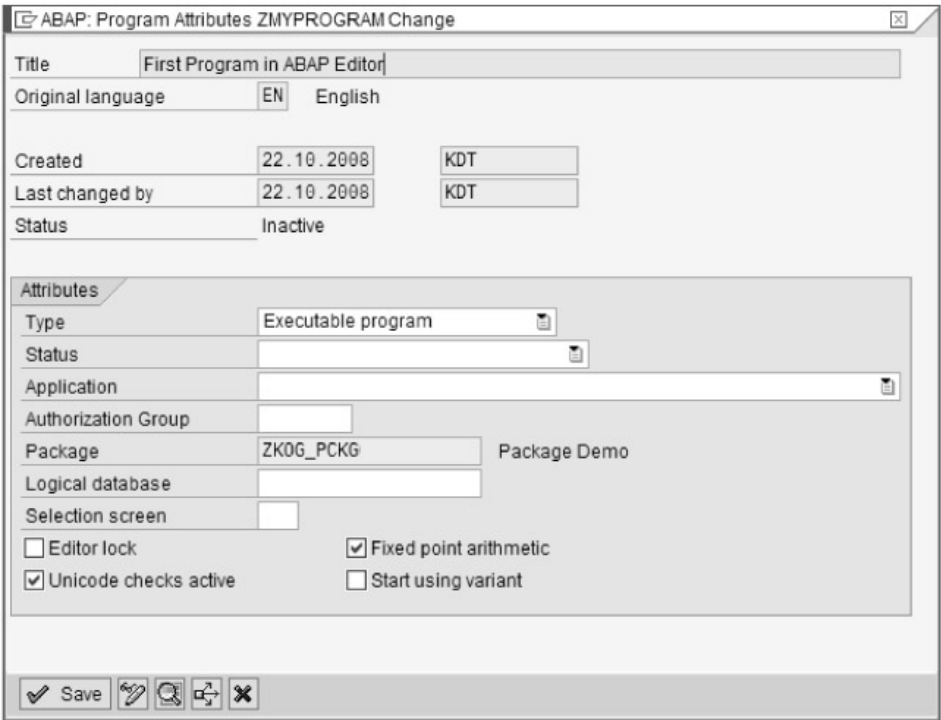
Table 6.2: Options provided in the subobjects group box

Radio Button	Description
Source code	Navigates to the program editor, where the source code can be written or edited. The program editor contains options to process, check, and save ABAP programs, as well as help and display functions for the source code.
Variants	Navigates to the variant maintenance tool, where you can define the fixed values of the input fields on the selection screen of a report. In addition to this, you can also edit, display, save, copy, print, and delete variants.
Attributes	Navigates to the program attributes screen, where you can maintain important program attributes, such as program type, program class, database used, authorizations, and runtime parameters. These attributes must be defined before entering the program code in ABAP Editor.
Documentation	Specifies a description of a program, explaining what it does and when it is used. You can also describe prerequisites

	of running a program, such as tables that must be maintained and programs that must be run. Moreover, you can also specify the examples that explain the report functions and other possible settings, such as output, integration, and activities.
Text elements	Maintains the text elements, such as title and headers, selection texts, and numbered texts, of an ABAP program.

Note A variant is a set of values entered in a selection screen, which is often used in a program. When you use an ABAP program in which selection screens are defined, numerous input fields are displayed where you enter a set of values. If you use the same set of values for the same program again and again, you can create a variant of the values stored in the selection set to reuse in the selection screen. A variant is created by using the variant maintenance tool, which is accessed through ABAP Editor (SE38).

The **Create** button in the **ABAP Editor: Initial Screen** (see [Figure 6.2](#)) is used to create an ABAP program. The **Display** and **Change** buttons, however, are used to view or modify ABAP programs, respectively. When you enter the name of a new program in the **Program** field of **ABAP Editor: Initial Screen** and click the **Create** pushbutton, the **ABAP: Program Attributes** dialog box appears, as shown in [Figure 6.3](#):



© SAP AG. All rights reserved.

Figure 6.3: The ABAP—program attributes dialog box

[Figure 6.3](#) shows the **ABAP: Program Attributes** dialog box, where you specify the values for the attributes of an ABAP program. In the **ABAP: Program Attributes** dialog box, you must specify the values in the following fields:

- **Title**—Describes the function of a program; that is, a short description about the program.
- **Original Language**—Contains the logon language of a user. This field is populated automatically by the SAP system.
- **Type**—Specifies the type of program.
- **Status**—Specifies whether the program is an SAP Standard Production Program, System Program, or Customer Production Program.
- **Application**—Contains the application to be used in an ABAP program, such as Financial Accounting. This field is optional.
- **Authorization Group**—Specifies the name of a program group. A program group is a collection of programs used for authorization checks.

- **Logical Database**—Defines the logical database used by an executable program (report) to read data. This field is applicable only to executable programs.
- **Selection Screen**—Specifies the selection screen based on the selection criteria of the logical database, and the `PARAMETERS` and `SELECT-OPTIONS` statements used in an ABAP program. This field is also applicable to executable programs.

The `ABAP: Program Attributes` dialog box also contains some check boxes, which implement additional features of `ABAP Editor` in an ABAP program. These check boxes are:

- **Editor lock**—Prevents other users from changing the attributes, text elements, and documentation of an ABAP program. Note that only the user who has worked on the program most recently can release the lock by clearing this check box.
- **Fixed point arithmetic**—Rounds off the value of the type `P` field according to the number of decimal places specified.
- **Start using variant**—Allows other users to start your program by using a variant. This field is also applicable to executable programs.
- **Unicode checks active**—Checks the program in the unicode format.

Note You can modify the values of the attributes of an ABAP program by using the `ABAP: Program Attributes` dialog box again. However, you navigate to the `ABAP: Program Attributes` dialog box by selecting the `Attributes` radio button in the `Subobjects` group box and clicking the `Change` button.

Now, let's learn about the different types of ABAP programs.

Types of ABAP Programs

Each ABAP program has a program type that determines whether a program can be executed. An ABAP program can be executed by either entering the program name in the initial screen of `ABAP Editor` or using a transaction code in the `Command` field. The `Type` field of `ABAP Editor` is used to specify the type of a program. The program types that can be selected are:

- Executable programs
- Module pools
- Subroutine pools
- Include programs

Besides the preceding program types, some other types of ABAP programs exist that are not created by `ABAP Editor`. Instead, `ABAP Workbench` provides specific tools to create and maintain these programs. These program types are:

- Function pools
- Class pools
- Interface pools

Now, let's discuss each program type in detail.

Defining Executable Programs

Executable programs are often called report programs because the source code of an executable program starts with the `REPORT` statement. These programs are created by processing the data stored in an SAP database. Executable programs cannot only retrieve the data from the database but also modify the data stored in the database. Executable programs can contain almost every kind of processing block, such as dialog modules, methods, and event blocks, except function modules and local classes. You do not need user-defined screens to control the executable programs. This is because the runtime environment of ABAP can call the processing blocks, screens, selection screens, and lists of executable programs automatically in a predefined sequence. Users can enter data on screens or selection screens, and then data is retrieved

from the database and processed accordingly. Finally, an output list is displayed on the basis of processed data.

Executable programs can be started by entering the program name or a transaction code. Note that if a transaction code is assigned to an executable program, the program can be started by using the transaction code instead of the program name. Moreover, executable programs can be linked to a logical database that contains subroutines. These subroutines are then called by a virtual system program in a predefined sequence.

Note Subroutines are used to make certain reporting functions reusable.

Defining Module Pools

A module pool is a program used to display data in and add functionality to a screen, such as a button, radio button, group box, or menu bar. You can write the screen flow logic using a module pool program; a subroutine is not used for this purpose. ABAP Editor is used to create module pool programs, which always start with the `PROGRAM` statement. A module pool is started using a transaction code, which is linked to a program and the initial screen of the program. Note that for a module pool program, you must define your own screens, including selection screens, by using Screen Painter. As a best practice, use a module pool program to write dialog-oriented programs using a large number of screens. Note that the flow logic of these screens determines the flow of the program.

Defining Subroutine Pools

A subroutine pool is a program that contains a collection of subroutines and local or global classes and interfaces that must be called externally in other ABAP programs. Prior to SAP release 6.0, a subroutine pool was used only to store and expose subroutines. However, with SAP release 6.0 and later, you can also call a subroutine pool program through the transaction code, which is attached with the public methods of local or global classes of this program. In ABAP Editor, a subroutine pool is created by using the `PROGRAM` statement.

Defining Include Programs

Include programs are not complete programs because they do not have a memory and cannot be executed, unlike other stand-alone programs. Alternatively, include programs act as a library of ABAP source code. Include programs are used to organize a program code into small units, which can be inserted in other ABAP programs. In other words, include programs are code snippets that can be reused in other programs. You can use these reusable code snippets with the help of the `INCLUDE` statement.

Defining Function Pools

Function pools, called function groups, are programs that contain function modules. A function pool program is created by using the `FUNCTION-POOL` statement. Each function pool contains global data, such as data objects, subroutines, or screens, which are shared by all the function modules of the function pool. Function modules are special ABAP procedures or routines that can be called in any ABAP program. Function Builder, a tool of ABAP Workbench, is used to create and manage the function pools (i.e., function groups) and function modules.

Defining Class Pools

A class pool is a special ABAP program used to store global classes and interfaces. All the ABAP programs can access these global classes and interfaces. A class pool is created by using the `CLASS-POOL` statement or by using `Class Builder`, a tool provided by ABAP Workbench. The class pool type of programs do not contain any screens or processing blocks. However, class pools contain methods that can be executed to implement the program logic.

Defining Interface Pools

Interface pools are programs that store global interfaces. An interface pool acts as a container that can store exactly one global interface. You can use an interface pool to implement the methods, which are defined in an interface of a class. To perform this, you must create reference variables of the interface type. Moreover, interface pools are maintained by using the `Class Builder` tool of ABAP Workbench.

Now, let's discuss how to write an ABAP program by following the syntax conventions.

ABAP Syntax

Every statement provided in the ABAP programming language has a predined syntax associated with it. Apart from this syntax, certain conventions need to be followed while writing the syntax for the statements. [Table 6.3](#) describes these conventions:

Table 6.3: Description of conventions

Term	Description
Statement	Specifies that ABAP statements and clauses are given in uppercase. It is not mandatory to use the ABAP keywords in uppercase; for instance, the WRITE keyword also can be Write or write.
Operand	Specifies that the operands used in the statement are in lowercase.
[]	Specifies that the element enclosed in the square brackets is optional.
	Specifies that there are elements on both sides of this symbol, and you can use either element in a program.
()	Specifies that parentheses is a part of the syntax of a statement and must be used while using the statement.
,	Separates one or more variables that you need to specify in a statement.
f1, f2	Represents variables indicated with indices. You can list as many variables in a program as you want.
.....	Represents the rest of the code in a program besides the syntax of ABAP statements.

Inserting Comments into ABAP Programs

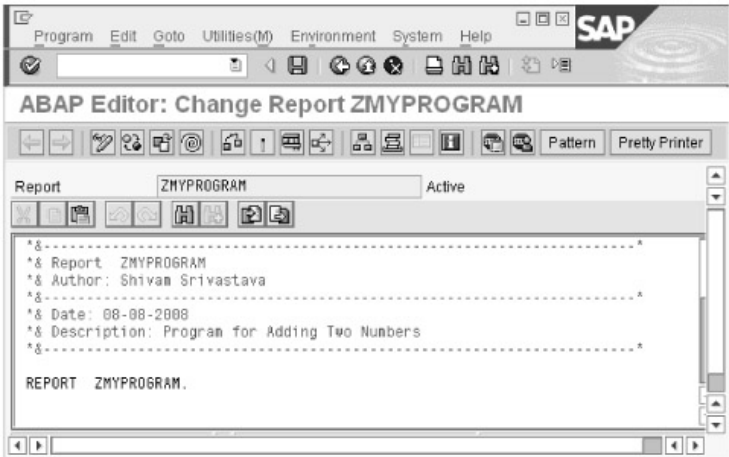
Comments are texts written between the statements of ABAP programs to document the program and to remind programmers about the purpose of using various statements. In an ABAP program, you can use the following ways to insert comments:

- **Line Comment**— Insert an asterisk at the leftmost column of a line; the entire line becomes a comment and is therefore not executed.
- **Partial Comment**— Insert a double quotation mark in the middle of a line of code so that everything to the right of the quote mark in the line of code becomes a comment.

Note You can convert a block of ABAP statements into comments by pressing the CTRL + < key combination. In addition, you can delete comments by first selecting the block and then pressing the CTRL + > key combination.

As a best practice, always include some comments, such as the date on which the program is created, description of the program created, and the name of the author, at the beginning of a program.

Figure 6.4 displays the comments included at the start of the ZMYPROGRAM program:



© SAP AG. All rights reserved.

Figure 6.4: Commenting the entire line

The comments in Figure 6.4 give information about the name of the author, the date on which the program is created, and the description related to the program.

Figure 6.5 shows how line comments, partial comments, and block comments can be inserted in a program:


```
*&-----*
*& Report  ZMYPROGRAM
*& Author:  Shivan Srivastava
*&-----*
*& Date:   08-08-2008
*& Description: Program for Adding Two Numbers
*&-----*

REPORT  ZMYPROGRAM.

DATA: a TYPE i,
      b TYPE i,
      c TYPE i.

* a = 10.
b = 20.
a = 30.

c = a + b. * Variable Containing the Value of Addition

*WRITE: / 'Value of a is:', a.
*WRITE: / 'Value of b is:', b.
*WRITE: / 'Value of c is:', c.
```

© SAP AG. All rights reserved.

Figure 6.5: Commenting partial code of an ABAP program

In Figure 6.5, the value 10 assigned to the variable a is ignored by the SAP system, because it is commented by using an asterisk. Note that the lines commented are ignored by the SAP system whenever the ZMYPROGRAM is executed.

Exploring Types and Objects

A type gives description of the technical attributes of the objects associated with that type. The objects are instances of types. Types and objects used in ABAP form a hierarchy as shown in Figure 6.6:

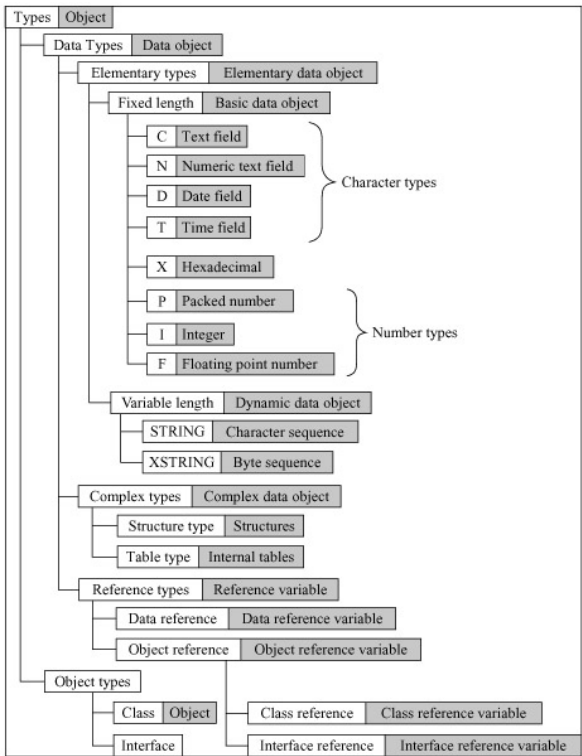


Figure 6.6: Hierarchy of ABAP types and ABAP objects

In Figure 6.6, notice that the types are categorized as data types and object types.

Data Types

Data types describe data objects. ABAP provides predefined data types that can be used in an ABAP program. Apart from the predefined data types, data types can be customized either locally in a program or globally in the SAP R/3 Repository.

The following are three kinds of data types:

- Elementary types
- Complex types
- Reference types

Describing Elementary Types

Elementary types are the smallest undivided unit of predefined data types. These elementary types are already defined in the SAP system and are visible in all ABAP programs. ABAP programs use these predefined elementary types to define local data types (structures) and data objects (fields) used in a program.

Note You can also create your own elementary data types with the help of the `TYPES` statement.

The following are two types of predefined elementary data types, which can be used in an ABAP program:

- Fixed-length
- Variable-length

These types are not derived from other types. The main difference between the fixed-and variable-length types is that the memory space required by the data objects of variable-length data types can change dynamically at runtime.

Now, let's discuss each of these data types in detail.

The Fixed Length Data Type

As the name suggests, the length of the fixed length data types are fixed at runtime. The fixed-length data types are:

- Character types
- Hexadecimal types
- Numeric types

Table 6.4 shows the values of various parameters related to fixed-length data types:

Table 6.4: Fixed-length data types

Data Type	Initial Field Length (in bytes)	Valid Field Length (in bytes)	Initial Value	Description
Numeric Types				
I	4	4	0	Specifies an integer (whole number)
F	8	8	0	Represents a floating point number
P	8	1–16	0	Specifies a packed number
Character Types				
C	1	1-65535	'.....'	Denotes a text field (alphanumeric characters)
D	8	8	'00000000'	Specifies a date field (Format:YYYYMMDD)
N	1	1-65535	'0....0'	Specifies a numeric text field (numeric characters)
T	6	6	'000000'	Specifies a time field (Format:HHMMSS)
Hexadecimal Types				
X	1	1-65535	X'0.....0'	Specifies a hexadecimal field

The Variable-Length Data Type

A variable-length data type is used for data objects whose length cannot be fixed, as the length varies according to the specified data. The variable-length data type is of two types, `STRING` and `XSTRING` described in [Table 6.5](#).

Table 6.5: Variable-length data types

Variable Length Data Type	Description
STRING	Specifies a sequence of characters with variable lengths. A string can contain any number of alphanumeric characters. The length of the string can be calculated by multiplying the number of characters with the length required for the internal representation of a single character.
XSTRING	Represents a string used for byte strings of a hexadecimal type. A byte string has a variable string and can contain any number of bytes. The length of a byte string is same as the number of bytes.

Describing Complex Types

A complex type is a combination of other types. There are no predefined complex data types in ABAP. However, complex types are created by users either in an ABAP program or in ABAP Dictionary. Complex types allow you to manage and process semantically related data under a common name. Complex types include the following:

- Structure types
- Table types

Now, let's discuss each of these types in detail.

Structure Types

A structure, also called a structure type, is a sequence of other data types defined in ABAP Dictionary, such as elementary types, structures, table types, and database tables. A structure type comprises a collection of components (also called fields), where each component has a name and data type. When you create a structure in your program, a component of the structure can refer to an elementary type, another structure, or a table type. In addition, you can nest a structure up to any level. You can use the `TYPE` clause in an ABAP program to refer to a structure directly. Moreover, structures can be used to define the data for screens and parameter types in function modules.

Table Types

A table type is another kind of complex data type that helps create complex data objects, such as internal tables. Internal tables consist of a series of lines, where all the lines have the same data type. You use internal tables when you need to use structured data within a program. [Table 6.6](#) shows the parameters according to which internal tables are characterized:

Table 6.6: Parameters of an internal table

Parameter	Description
Line or row type	Specifies that a row of an internal table can be of the elementary type, complex type, or reference type.
Key	Specifies a field or a group of fields as a key of an internal table that identifies the table rows. A key contains the fields of elementary types. Keys can be of two types, unique and non-unique.
Access method	Describes how ABAP programs access individual table entries. The three types of access are unsorted types, sorted index tables, and hashed tables.

Describing Reference Data Types

Reference types are data types used to describe data objects containing references (pointers) to other objects, such as data objects and objects created in the ABAP Objects language. To use references, you need to define the references in your program.

Object Types

The objects in ABAP Objects (object-oriented extensions of the ABAP language) are described with the help of object types. These object types are as follows:

- **Classes**— Contains the description of an object. In addition, a class defines the data types and functions that an object contains.
- **Interfaces**— Contains the description of an object, similar to a class. However, interfaces partially describe the aspects of an object. An interface contains several data types and functions that can be used in classes.
- **Functions**— Describes the behavior of an object. Functions can access any attributes of a class.

Note To learn more about ABAP Objects, refer to Appendix A.

All the other data types, as shown in [Figure 6.4](#), can be used in ABAP Objects. Depending on your needs, object types are declared in a program or in the SAP R/3 Repository.

Now, let's explore the two types of objects that can be created from ABAP types.

Objects in ABAP

The objects in ABAP are created with the help of ABAP data types (see [Figure 6.6](#)).

Describing Data Objects

Data objects are fields that hold the data used by ABAP programs at runtime. Data objects exist only until a program is being executed. For example, if a user wants to read data from a database table or a sequential file, the data must first be read into the data object. The following are types kinds of data objects:

- **Literal**— Specifies an unnamed data object in a program having a fixed character string or a number. Literals can be a character or a numeric type, such as 45, 'Hello', '238', and 68.92. They are not created by declarative statements but have fixed technical attributes, such as field length, number of decimal places, and data type.
- **Named data object**— Specifies a named data object that is declared either statically or dynamically at runtime. Similar to a literal, the technical attributes of named data objects, such as length, number of decimal places, and the data type, are always fixed. A named data object is assigned a name with which you can address the data object from ABAP programs. Text symbols, variables, constants, and interface work areas are examples of named data objects.
- **Predefined data object**— Specifies an already available data object in the SAP system at runtime. A predefined data object does not need to be declared explicitly.
- **Dynamic or anonymous data object**— Specifies an unnamed data object, which is created dynamically in a program and used with a data reference variable.

Note A dynamic data object is created during the execution of a program. It is referenced by a data reference variable and the following syntax of the `CREATE DATA` statement:

```
CREATE DATA <datarefvar> {TYPE data_type} | {LIKE
data_object}.
```

In the preceding syntax, <datarefvar> represents a data reference variable that points to a data object created in the internal session of the current ABAP program. Note that the data object does not have its own name. You can refer this data object by using the <datarefvar> data reference variable.

Describing Objects

An ABAP Object is an instance of a class; for example, Car is a class and Honda City is its object. Similar to other programming languages, ABAP Objects also provide the basic features of object-oriented programming; that is, encapsulation, inheritance, and polymorphism. An ABAP Object contains methods, events, and data. A class can contain numerous objects; however, each object has its own identity and attributes.

Variables in ABAP

Variables are named data objects used to store values within the allotted memory area of a program. As the name suggests, users can change the content of variables with the help of ABAP statements. [Table 6.7](#) shows the statements used to declare a variable statically:

Table 6.7: Statements used to declare variables statically

Statement	Description
DATA	Declares the variable whose lifetime is linked to the context of the declaration.
STATICS	Declares the variables that can be used in subroutines, function modules, and static methods.
CLASS-DATA	Declares variables within the classes.
PARAMETERS	Declares the elementary data objects that are linked to input fields on a selection screen.
SELECT-OPTIONS	Declares the internal tables that are linked to input fields on a selection screen.
RANGES	Declares internal tables with the same structure as defined in the <code>SELECT-OPTIONS</code> statement, provided the declared internal table is not linked to a selection screen.

Apart from declaring the variables statically, you can also declare the variables dynamically; that is, whenever you call procedures, such as subroutines. For example, the variables declared in the `FORM` statement (also known as formal parameters) inherit the technical characteristics, such as the data type and length of the parameters defined with the `PERFORM` statement (also known as actual parameters). In addition to this, the values of the actual parameters are assigned to the formal parameters.

Now, let's discuss the `DATA` and the `PARAMETERS` statements in detail.

The `DATA` Statement

The `DATA` statement is used to declare variables in an ABAP program. Variables defined by the `DATA` statement have a predefined or user-defined data type. The following syntax shows how to use the `DATA` statement:

```
DATA <f> ... [TYPE <type>] [LIKE <obj>]... [VALUE <val>].
```

Table 6.8 describes the clauses that can be used in the `DATA` statement:

Table 6.8: Description of the clauses of the `DATA` statement

Clause	Description
<f>	Specifies the name of a variable. The name of the variable can be up to 30 characters long.
TYPE <type>	Specifies that the type of <f> variable. Any data type with fully specified technical attributes is known as <type>. The possible <type> allotted to a variable can be a nongeneric predefined ABAP type (D, F, I, T, STRING, XSTRING) or any existing local data type in a program (the <code>TYPES</code> statement is used to define local data types in a program) or an ABAP Dictionary data type.
LIKE <obj>	Shows that the declared variable name <f> inherits the same technical attributes as an existing data object, <obj>. Predefined data objects that do not need to be declared separately are represented by <obj>.
VALUE <val>	Specifies the initial value of the <f> variable. In case you define an elementary fixed-length variable, the <code>DATA</code> statement automatically populates the value of the variable with the type-specific initial value, as listed in Table 6.5. Other possible values for <val> can be a literal, constant, or an explicit clause, such as <code>IS INITIAL</code> .

The following conventions are used while naming a variable in an ABAP program:

- You cannot use special characters such as "+" and "," to name variables.
- The name of the predefined data objects cannot be changed.
- The name of the variable cannot be the same as any ABAP keyword or clause.
- The name of the variables must convey the meaning of the variable without the need for further comments.
- Hyphens are reserved to represent the components of structures. Therefore, avoid using hyphens in variable names.
- The underscore character can be used to separate compound words.

The following code snippet shows how to define variables by using the `DATA` statement:

```
DATA d1(2) TYPE C.
DATA d2 LIKE d1.
DATA min_value TYPE I VALUE 10.
```

In the previous code snippet, `d1` is a variable of `C` type, `d2` is a variable of `d1` type, and `min_value` is a variable of `I` type. The following code snippet demonstrates another way to declare variables:

```
*-----*
*/ Structure Declarations
*/
TYPES: BEGIN OF number,
        Number_1 TYPE I,
        Number_2 TYPE p DECIMALS 2,
        End of number.
*-----*
*/Data Declarations
*/
DATA:      n_number TYPE number,
        Num LIKE n_number-Number_1,
        Date LIKE SY-DATUM,
        Year TYPE i.
*-----*
```

In the preceding code snippet, variables are declared with reference to the internal type, named `number`, in a program. Another variable, `Num`, is declared similar to the component of the existing data object `n_number`. The third variable, `Date`, refers to the `SY-DATUM` variable and the `Year` variable has a reference to the predefined ABAP type `I`.

Listing 6.1 shows the incorrect access of a variable defined with the help of the `DATA` statement:

Listing 6.1: Accessing a variable incorrectly

```
Report ZNAMEDISPLAY.
*/ accessing a variable
DATA name1(7) VALUE 'shivam'.
WRITE: name1, name2.
DATA name2(10) value 'srivastava'.
```

In Listing 6.1, notice that the `name2` variable accesses the `WRITE` statement before the definition of the variable. On checking Listing 6.1 in ABAP Editor, a syntax error is generated, as shown in Figure 6.7:



© SAP AG. All rights reserved.

Figure 6.7: Showing a syntax error in the SAP system

In [Figure 6.7](#), notice the syntactical error, which can be removed by moving the definition of the `name2` variable before the `WRITE` statement, that is accessing it.

Apart from the `DATA` statement, you can use the `PARAMETERS` statement to define the variables (or parameters) in an ABAP program. Now, let's discuss the `PARAMETERS` statement in detail.

The `PARAMETERS` Statement

The `PARAMETERS` statement is used to enter the values of the variables on both the standard selection screen as well as user-defined selection screens. These variables (also known as parameters) are defined with the help of the `PARAMETERS` statement. Each parameter defined with the `PARAMETERS` statement appears as an input field on the relevant selection screen. The `PARAMETERS` statement is used to enter single values in the input fields. The flow of the program can be controlled with the help of parameters. The syntax of the `PARAMETERS` statement is

```
PARAMETERS <p> [(<length>)] [TYPE <type>|LIKE <obj>]
[DECIMALS <d>].
```

In the preceding syntax, the use of the `TYPE` or `LIKE` clause is optional, but the data must be of the `C` type with length 1.

[Table 6.9](#) describes the clauses used in the `PARAMETERS` statement:

Table 6.9: Clauses of the `PARAMETERS` statement

Clause	Description
<p>	Represents the name of a parameter. The maximum length for naming parameters is 8 instead of 30, as in the <code>DATA</code> statement.
<length>	Specifies the length of a variable.
TYPE <type>	Specifies a data type with fully specified technical attributes. The possible <type> allotted to a variable can be a predefined ABAP type (<code>D</code> , <code>I</code> , <code>T</code> , <code>STRING</code> , <code>XSTRING</code>) or an ABAP Dictionary data type. The <code>TYPE</code> keyword shows that the declared parameter name <p> inherits the same technical attributes as that of an existing data type <type>.
LIKE <obj>	Specifies a predefined data object that is already present and need not be declared separately, such as system variables. Fields used to provide information about the current state of the SAP system are known as system variables. The <code>LIKE</code> keyword specifies that the declared parameter name <p> inherits the same technical attributes as that of the existing data object <obj>.
DECIMALS <d>	Specifies the number of decimal places for numeric values.

Note The `TYPE` and `LIKE` clauses are used in ABAP statements for various purposes, some of which are:

- Defining local types in a program
- Declaring data objects
- Creating dynamic data objects
- Specifying the type of formal parameters in subroutines
- Specifying the type of formal parameters in methods
- Specifying the type of field symbols

[Listing 6.2](#) shows how to declare variable using the `PARAMETERS` statement:

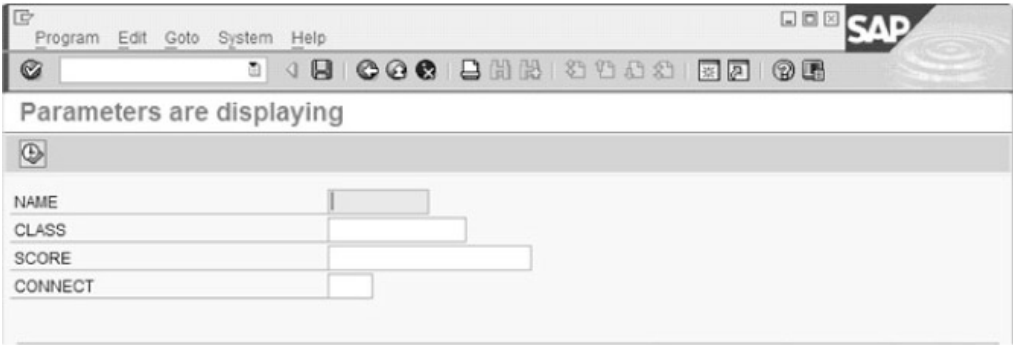
Listing 6.2: Using the `PARAMETERS` statement

```
REPORT ZPARAMETERDISPLAY.
*-----*
*/Creating Parameters
*/
PARAMETERS: NAME(10) TYPE C,
CLASS TYPE I,
SCORE TYPE P DECIMALS 2,
CONNECT TYPE MARA-MATNR.
```

In Listing 6.2, four parameters are created and displayed on the standard selection screen. The parameters are:

- NAME—Represents a parameter of 10 characters
- CLASS—Specifies a parameter of integer type with the default size in bytes
- SCORE—Represents a packed type parameter, with values up to two decimal places
- CONNECT—Refers to the MARA-MATNR type of ABAP Dictionary

Figure 6.8 shows the output of Listing 6.2:



© SAP AG. All rights reserved.

Figure 6.8: Output of the PARAMETERS statement

Apart from the clauses shown in Table 6.10, you can also define check boxes and radio buttons on the standard selection screen by using the PARAMETERS statement.

Table 6.10: Description of clauses

Clause	Description
<f>	Specifies a name for the constant.
TYPE <type>	Represents a constant named <f>, which inherits the same technical attributes as the existing data type <type>. Any data type with fully specified technical attributes is known as a <type>.
LIKE <obj>	Specifies that the constant name <f> inherits the same technical attributes as an existing data object <obj>.
VALUE <val>	Assigns an initial value to the declared constant name <f>. If you define an elementary fixed-length variable, the CONSTANTS statement automatically populates the value of the variable with the type-specific initial value, as listed in Table 6.5. The other possible <val> values can be a literal, constant, or an explicit clause, such as IS INITIAL.

Defining Check Boxes by Using the PARAMETERS Statement

The PARAMETERS statement can also be used to define a check box on the standard selection screen. The following syntax is used to create a check box:

```
PARAMETERS <p> ..... AS CHECKBOX.....
```

The parameter field name <p> is used to create a check box. The field name has a C type and 1 as the length, by default. The ' ' and X are valid values used in place of the <p> parameter field.

Listing 6.3 shows how to display a field with a check box by using the PARAMETERS statement:

Listing 6.3: Creating check boxes

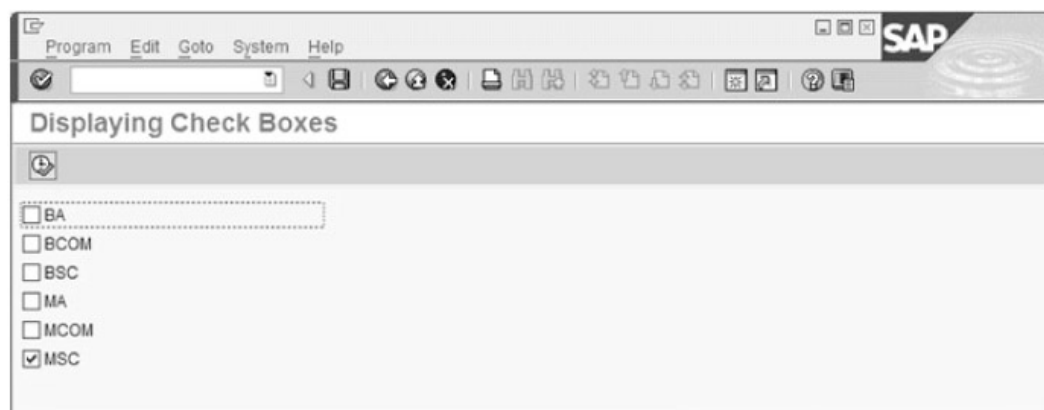
```
REPORT ZCHECKBOXDISPLAY.  
*-----*  
*/Creating Checkboxes Using PARAMETERS Statement
```

```

* /
PARAMETERS: BA AS CHECKBOX,
BCom AS CHECKBOX,
BSc AS CHECKBOX,
MA AS CHECKBOX,
MCom AS CHECKBOX,
MSc AS CHECKBOX DEFAULT 'X'.
*-----*

```

In [Listing 6.3](#), the `PARAMETERS` statement is used to create six check boxes: BA, BCom, BSc, MA, MCom, and MSc. The MSc check box is set as the default. [Figure 6.9](#) shows the output of [Listing 6.3](#):



© SAP AG. All rights reserved.

Figure 6.9: Check boxes on the standard selection screen

Defining Radio Buttons by Using the `PARAMETERS` Statement

The `PARAMETERS` statement is also used to define radio buttons on a standard selection screen. The syntax to create radio buttons is:

```
PARAMETERS <p> ..... RADIOBUTTON GROUP <radi>.....
```

In the preceding syntax, a parameter `<p>` is created with type `C` and length 1. The parameter `<p>` is also assigned to a group `<radi>`. The `RAD1` group is assigned with at least two parameters. Only one parameter per group can have the default value of `X`, assigned by using the `DEFAULT` clause. If the `DEFAULT` clause is not used, the first parameter of each group is set to zero. When the user clicks a radio button on the selection screen, the respective parameter is assigned the value `X`, while all the other parameters of the same group are assigned the value `'`. [Listing 6.4](#) shows the creation of radio buttons:

Listing 6.4: Creating radio buttons

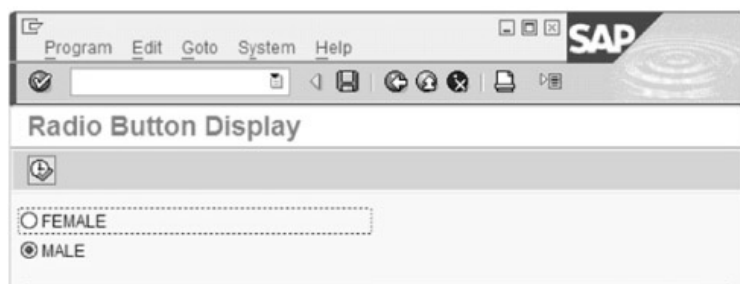
```

REPORT ZRADIO_BUTTON_DISPLAY.
*-----*
*/Creating Radiobuttons
* /
PARAMETERS: FEMALE RADIOBUTTON GROUP RAD1,
MALE RADIOBUTTON GROUP RAD1 DEFAULT 'X'.
*-----*

```

In [Listing 6.4](#), the `PARAMETERS` statement is used to create two radio buttons, FEMALE and MALE. The MALE radio button is set as the default.

[Figure 6.10](#) shows the output of [Listing 6.4](#):



© SAP AG. All rights reserved.

Figure 6.10: Radio buttons display

Constants in ABAP

Constants are named data objects created statically by using declarative statements. In a program, a constant is declared by assigning a value to it, which is stored in the program's memory area. The value assigned to a constant cannot be changed during the execution of the program. When the value of the constant is changed, a syntax or runtime error may occur. These named data objects are declared with the help of the `CONSTANTS` statement.

The syntax of the `CONSTANTS` statement is:

```
CONSTANTS <f> ... [TYPE <type>|LIKE <obj>]...
[VALUE <val>].
```

Notice that the syntax of the `CONSTANTS` statement is similar to the `DATA` statement.

Note You must use the `VALUE` clause in the `CONSTANTS` statement. The `VALUE` clause is used to assign an initial value to the constant during its declaration. This `VALUE` clause is optional with the `DATA` statement.

Constants cannot be defined for `STRINGS`, internal tables, references, and structures containing internal tables.

Table 6.10 describes the clauses used in the `CONSTANTS` statement:

The following code snippet shows how to define constants by using the `CONSTANTS` statement:

```
CONSTANTS: abc TYPE P DECIMALS 5 VALUE '1.23456',
           def TYPE C VALUE IS INITAIL.
```

In the preceding code snippet, you see constants that are declared with the `CONSTANTS` statement. The declared constants refer to elementary data types; therefore, the elementary data types are also called elementary constants.

The following code snippet shows how to define complex constants:

```
*-----*
*/Defining a complex constant
*/
CONSTANTS: BEGIN OF EMPLOYEE,
           Name(20) TYPE c VALUE 'SHIVAM SRIVASTAVA',
           Company(50) TYPE c VALUE 'Software Solutions
           Incorporation',
           City(10) TYPE c VALUE 'New Delhi',
           Pincode(8) TYPE i VALUE '110002',
           END OF EMPLOYEE.
*-----*
```

In the preceding code snippet, `EMPLOYEE` is a complex constant that is composed of the `Name`, `Company`, `City`, and `Pincode` fields.

The `TABLES` Statement

The `TABLES` statement is used to create a structure having the same name as a database table, view, or structure defined in ABAP Dictionary. This statement actually defines a table work area, which is a kind of interface work area, used to create in the shared area of a program.

Note An interface work area is a special named data object used to pass data between:

- Screens and ABAP programs
- Logical databases and ABAP programs
- ABAP programs and external subroutines

The syntax of the `TABLES` statement is:

```
TABLES <dbtab>.
```

In the preceding syntax, `<dbtab>` represents a structure with the same data type and name of a database table, a view, or a structure from ABAP Dictionary. Prior to SAP release 4.0, the `TABLES` statement was necessary to include a database table in an ABAP program. Nowadays, SAP systems use Open SQL statements, which do not require the `TABLES` statement in the program. However, the `TABLES` statement is still used to define input and output fields on a screen with reference to database tables, views, or structures.

Assignment Statements

Assignment statements are used to assign the values of data objects to a variable in an ABAP program. The four different types of assignment statements are:

- `MOVE`
- `MOVE-CORRESPONDING`
- `WRITE TO`
- `CLEAR`

Now, let's discuss each statement in detail.

The `MOVE` Statement

The `MOVE` statement is used to assign the value of a data object to another data object; that is, to transfer the content of one field to another. The following syntax shows how to use the `MOVE` statement:

```
MOVE <f1> TO <f2>.
```

In this syntax, `<f1>` is the data object whose data has to be transferred to another data object, `<f2>`. The equivalent statement for the `MOVE` statement is `<f2> = <f1>`. The `MOVE` statement assigns the value of one data object to another, but the value of the original data object remains unchanged. It is not necessary for `<f1>` to be a data object; it can be a literal, text symbol, or constant.

You can use the `MOVE` statement to assign values to multiple data objects, such as `<f4> = <f3> = <f2> = <f1>`. The following syntax shows how to use the `MOVE` statement for multiple assignments:

```
MOVE <f1> TO <f2>.  
MOVE <f2> TO <f3>.  
MOVE <f3> TO <f4>.
```

In this syntax, you see that the content of `<f1>` is transferred to `<f2>`, the content of `<f2>` is transferred to `<f3>`, and, finally, the content of `<f3>` is transferred to `<f4>`. To make the transfer of values possible, the data types and data objects must be compatible. The following points must be remembered regarding the compatibility of data types and data objects:

- If the data objects `<f1>` and `<f2>` are fully compatible (that is, their data types, field length, and the number of decimal places are the same), the content of the source field `<f1>` is transferred byte by byte into the target field `<f2>`, without any calculation. In such cases, the working of the `MOVE` statement is most efficient.
- If the data objects are incompatible (that is, if the two fields are of the same type but of different lengths), the content of the source field is converted so that the source field is compatible with the data type of the target field `<f2>`. This transfer happens only if a conversion rule exists between the data types `<f1>` and `<f2>`.
- If the data types are not compatible and no conversion rule exists, transfer of values does not take place.

In the case of `MOVE` statements, if values have to be transferred between noncompatible data objects, the value of the source object always is converted into the data type of the target object. The type of conversion performed in `MOVE` statements is valid for all the different kinds of value assignments.

Note If you try to assign values between two data types for which no conversion rule exists, a syntax error or runtime error occurs.

Table 6.11 shows the conversion rules that exist between different data types:

Table 6.11: Conversion rules

Type of Source Field	Type of Target Field	Conversion Rule
C	P	Specifies that the source field may contain numbers, a single decimal point, or an optional sign, which can be trailing or leading. In addition to this, blanks can appear on either side of the value. Any blank value is converted to zero.
C	D	Specifies that the source field should contain a valid date in the YYYYMMDD format. If the source field does not contain a valid date format, no error occurs; however, an invalid value is assigned to the target field.
C	T	Specifies that the source field should contain a valid time in the HHMMSS format. If the source field does not contain a valid time format, no error occurs; however, an invalid value is assigned to the target field.
C	N	Specifies that the source field is scanned from left to right and only the digits 0–9 are transferred to the target field (right-justified). In addition, values on the left are padded with zeroes.
C	X	Specifies that the source field contains a hexadecimal-character string and that the valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Note that this character string is packed as a hexadecimal number padded with zeroes or truncated on the right.
P	C	Specifies that the value in the target field is right-justified in the target field, with the rightmost byte reserved for the trailing sign.
P	D	Specifies that the value in the source field is interpreted as the number of days and stored internally in the YYYYMMDD format.
P	T	Specifies that the value in the source field is interpreted as the number of seconds since midnight converted to 24-hour clock time, and stored internally in the HHMMSS format.
D	P	Specifies that the value in the source field is converted to a number representing the number of days.
T	P	Specifies that the value in the source field is converted to a number representing the number of seconds since midnight.

Listing 6.5 shows the use of the `MOVE` statement:

Listing 6.5: Using the MOVE statement

```

REPORT ZMOVE_DEMO.
*-----*
*/ Data Declarations
*/
DATA: Number_1(10) TYPE c,
      Number_2 TYPE p DECIMALS 2,
      Number_3 TYPE i.
*-----*
Number_1 = 100.
*-----**
/MOVE statement */
MOVE '5.75' TO Number_2.

*-----*
Number_3 = Number_1.
WRITE: 'Number_1 =',Number_1.
WRITE: / 'Number_2 =',Number_2.
WRITE: / 'Number_3 =',Number_3.

```


In [Listing 6.5](#), Number_1, Number_2, and Number_3 are data objects of data type C, P, and I, respectively. The length of the Number_1 data object is 10. Number_1 data object stores 100, Number_2 data object stores 5.75, and Number_3 data object stores the value of Number_1 data object that is, 100. Note that the values in the Number_1 and Number_3 data objects are assigned directly, while the value of the Number_2 data object is assigned by using the MOVE statement.:

[Figure 6.11](#) shows the output of [Listing 6.5](#):

```
Number_1 =      100
Number_2 =      5.75
Number_3 =      100
```

© SAP AG. All rights reserved.

Figure 6.11: Output of the MOVE statement

The MOVE-CORRESPONDING Statement

The MOVE-CORRESPONDING statement is used to assign values between the components of two or more structures. The syntax of the MOVE-CORRESPONDING statement is

```
MOVE-CORRESPONDING <struct1> TO <struct2>.
```

When you execute the MOVE-CORRESPONDING statement, the content of the components of the <struct1> structure is copied to the components of the structure <struct2>, which contains identical names. In addition to this, the syntax is actually broken down into a set of MOVE statements, one for each pair of fields which have identical names. The following is the equivalent syntax of the MOVE-CORRESPONDING statement:

```
MOVE STRUCT1-<Ci> TO STRUCT-<Ci>.
```

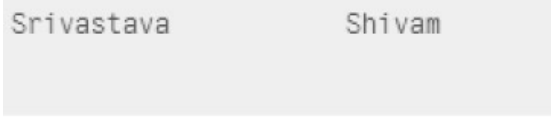
[Listing 6.6](#) shows the working of the MOVE-CORRESPONDING statement.

Listing 6.6: Using the MOVE-CORRESPONDING statement

```
REPORT ZMOVE_DEMO.
*-----*
*/ Data Declarations
*/
DATA: BEGIN OF ADDRESS,
      FNAME(20) TYPE c VALUE 'Shivam',
      LNAME(20) TYPE c VALUE 'Srivastava',
      CITY(20) TYPE c VALUE 'Lucknow',
      END OF ADDRESS.
*-----*
*/ Data Declarations
*/
DATA: BEGIN OF NAME,
      LNAME(20) TYPE c,
      FNAME(20) TYPE c,
      END OF NAME.
*-----**/ MOVE-CORRESPONDING statement
MOVE-CORRESPONDING Address TO Name.
*-----*
WRITE:/ NAME-LNAME,
      NAME-FNAME.
```

[Listing 6.6](#) shows that the values of NAME-LNAME and NAME-FNAME are set to SRIVASTAVA and SHIVAM, respectively.

[Figure 6.12](#) shows the output of [Listing 6.6](#):



© SAP AG. All rights reserved.

Figure 6.12: Output of the MOVE-CORRESPONDING statement**The WRITE TO Statement**

The WRITE TO statement is an assignment statement that converts the content of the source field into a field of type C. The syntax of the WRITE TO statement is

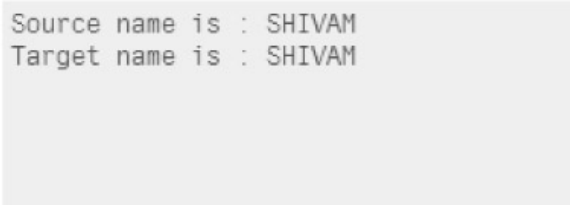
```
WRITE <f1> TO <f2> [<option>].
```

In the preceding syntax, the WRITE TO statement converts the content of the data object <f1> to type C and places the string into the <f2> variable. Listing 6.7 shows the use of the WRITE TO statement:

Listing 6.7: Using the WRITE TO statement

```
REPORT ZWRITE_DEMO.
*-----*
*/ Data Declarations
*/
DATA: Name(10) TYPE C VALUE 'SHIVAM',
      TEXT(10).
*-----*
*/Using WRITE TO statement to show source and target names
*/
WRITE : 'Source name is : ', Name.WRITE: Name TO TEXT.
WRITE: / 'Target name is : ', TEXT.
*-----*
```

In Listing 6.7, the Name variable is of type C. The WRITE TO statement assigns the content of the source structure, that is, Name, to the target structure, that is, TEXT. The WRITE TO statement displays the content of both the source and target structure. Figure 6.13 shows the source name and the target name as SHIVAM:



© SAP AG. All rights reserved.

Figure 6.13: Output of the WRITE TO statement

Note The data type of <f1> must be convertible into a character field. If <f1> is not convertible into a character field, a syntax or runtime error occurs. The content of the data object <f1> remains unchanged. The content of the <f2> data object always is regarded as a character string. Therefore, you must not use a target field with a numeric data type (F, I, or P). If you use a numeric target field, the SAP system displays a syntax error.

The CLEAR Statement

The CLEAR statement is used to reset the value of a data object. Resetting the values of the data objects depends on the data type of the data objects. For example, data objects of type I are set to zero and data objects of type C are set to null. The following is the syntax of the CLEAR statement:

```
CLEAR <f>.
```

In this syntax, <f> is either a field or a field string. If <f> is a field, the CLEAR statement resets the value of the field to its initial value. The initial value of the field depends on the data type of the field. If <f> is a field string, the CLEAR statement resets each of the individual fields in the header line of the field string to their respective initial values:

Table 6.12 shows the result of the CLEAR statement for the data object of the different data types:

Table 6.12: Impact of the CLEAR statement on different data types

Data Type	Description of CLEAR Statement Impact
Elementary Data Type	Resets the values to initial values and not to the start value, which is set using the VALUE parameter of the DATA statement.
Reference	Resets a reference variable to its initial value so that it does not point to any object.
Structure	Resets the individual components of a structure to their respective initial values.
Internal Table	Deletes the entire content of the internal table.

Note You cannot use the CLEAR statement to reset a constant.

Listing 6.8 shows the working of the CLEAR statement:

Listing 6.8: The CLEAR Statement

```
REPORT ZCLEAR.
*-----*
*/ Data Declarations
*/
DATA number TYPE I VALUE '80'.
*-----*
WRITE number.
*-----**
/Using CLEAR statement
CLEAR number.
WRITE / number.
*-----*
```

In Listing 6.8, the CLEAR statement resets the content of a field from 80 to its initial value, 0. Figure 6.14 shows the output of Listing 6.8:



© SAP AG. All rights reserved.

Figure 6.14: Output of the CLEAR statement

Formatting Options

ABAP offers various types of formatting options to format the output of programs. For example, you can create a list that includes various items in different colors or formatting styles. ABAP offers the following statements to perform the formatting:

- The WRITE statement
- The FORMAT statement

Now, let's discuss each of these statements in detail.

The WRITE Statement

The WRITE statement is a formatting statement used to display data on a screen. There are different formatting options for the WRITE statement. The syntax of the WRITE statement is:

```
WRITE <format> <f> <options>.
```

In the preceding syntax, <format> represents the output format specification, which can be a forward slash (/) that

indicates display of the output starting from a new line. In addition to the forward slash, the format specification includes a column number and column length. For example, the `WRITE/04(6)` statement shows that a new line begins with column 4 and the column length is 6, whereas the `WRITE 20` statement shows the current line with column 20. The `<f>` parameter can represent a data variable, text literal, or numbered text.

Table 6.13 describes various clauses used in the `WRITE` statement for formatting:

Table 6.13: Clauses representing various formatting options for all data types

Clause	Description
LEFT-JUSTIFIED	Specifies that the output is left-justified.
CENTERED	Specifies that the output is centered.
RIGHT-JUSTIFIED	Specifies that the output is right-justified.
UNDER <g>	Specifies that the output starts directly under field <g>.
NO-GAP	Specifies that the blank after field <f> is rejected.
USING EDIT MASK <m>	Specifies the specification of the format template <m>.
USING NO EDIT MASK	Specifies that the format template specified in the ABAP Dictionary is deactivated.
NO-ZERO	Specifies that if a field contains only zeroes, they are replaced by blanks.

Table 6.14 shows the formatting options for numeric type fields only:

Table 6.14: Formatting options for numeric type fields

Formatting Option	Function
NO-SIGN	Specifies that no leading sign is displayed on the screen.
EXPONENT <e>	Specifies that in type <code>F</code> , fields (i.e., floating point type fields), the exponent is defined in <e>.
ROUND <r>	Specifies that type <code>P</code> fields (i.e., packed numeric type fields) are first multiplied by $10^{**(-r)}$ and then rounded off to an integer value.
CURRENCY <c>	Specifies that the formatting is done according to the value that is stored in the <code>TCURX</code> currency <c> database table.
UNIT <u>	Specifies that the number of decimal places is fixed according to the <u> unit as specified in the <code>T006</code> database table for type <code>P</code> , (that is, packed type fields).
DECIMALS <d>	Specifies that the number of digits <d> must be displayed after the decimal point.

Table 6.15 lists different formatting options for the date fields and examples:

Table 6.15: Different formatting options for date fields and examples

Formatting	Example
DD/MM/YY	13/01/85
MM/DD/YY	01/13/85
DD/MM/YYYY	13/01/1985
MM/DD/YYYY	01/13/1985
DDMMYY	130185
MMDDYY	011385
YYMMDD	850113

Note In Table 6.15, all the examples are given in the context of the date January 13, 1985. Here, DD stands for the date in two figures, MM stands for the month in two figures, YY stands for the year in two figures, and YYYY stands for the year in four figures.

Table 6.16 shows some examples of ABAP code that implements the formatting options along with the output:

Table 6.16: Examples of formatting options and their output

ABAP Coding	Screen Output
DATA: g(6) TYPE c VALUE 'Shivam',	Shivam Srivastava
f(10) TYPE c VALUE 'Srivastava'.	Shivam
WRITE: g, f.	Srivastava
WRITE:/10 g,	ShivamSrivastava
/ f UNDER g.	
WRITE:/ g NO-GAP, f.	
DATA time TYPE t VALUE '123456'.	
WRITE: time,	123456
/(8) time USING EDIT MASK '__:__:__'.	12:34:56
WRITE: '000321',	000321
/ '000321' NO-ZERO.	321
DATA float TYPE f VALUE '12345	
6789.0'.	
WRITE float EXPONENT 3.	123456.789000000000E+03
DATA pack TYPE p VALUE	
'123.456'DECIMALS 3.	123.46
WRITE pack DECIMALS 2.	12,345.600
WRITE: /pack ROUND -2,	1,234.560
/ pack ROUND -1,	12.346
/ pack ROUND 1,	1.235
/ packROUND 2.	
WRITE: SY-DATUM,	27.06.1995
/ SY-DATUMYYMMDD.	950627

The **FORMAT** Statement

Different types of syntaxes are available for specific actions performed with the **FORMAT** statement. Table 6.17 shows the different variations in these syntaxes:

Table 6.17: FORMAT statement syntaxes

FORMAT Statement	Description
FORMAT <option1> [ON OFF] <option2> [ON OFF].....	Specifies that the formatting options are set statically in a program. The option mentioned in the <option> expression is applicable to all output until the formatting option is turned off by using the OFF clause, as indicated in the statement. Note that the ON or OFF clause is optional.
FORMAT <option1> = <var1> <option2> = <var2>....	Specifies that the formatting options are set dynamically at runtime. The variables var1, var2, are interpreted as numbers and should be declared with the data type as I. If the content in the var1 and var2 variables is zero, the variable has the same effect as the OFF clause. If the numbers in the var1 and var2 variables are not equal to zero, either the color is shown according to the numbers stored in the variables or the variables have the same effect as that of the ON clause. Note that the formatting options are all set to their default values whenever they are used for the new event.
FORMAT RESET	Sets all the formatting options to off.

Using the **FORMAT** Statement Clauses

The **FORMAT** statement is also used to display the output in a colored format. The following syntax is used to set colors in a program:

```
FORMAT COLOR <num> [ON] INTENSIFIED [ON|OFF] INVERSE [ON|OFF].
```

The following syntax is used to set colors at runtime:

```
FORMAT COLOR = <const> INTENSIFIED = <int> INVERSE = <inv>.
```

In both syntaxes, the `COLOR` clause is used to set the background color of a line. In addition, the use of the `INVERSE ON` clause allows the SAP system to change the foreground color of a line instead of the background color. The `<num>` expression represents a color number or color specification. [Table 6.18](#) lists the different possible values of the `<num>` and `<const>` expressions:

Table 6.18: Values of the `<num>` and `<const>` expressions

Color Number for the <code><num></code> Expression	Color Specification for the <code><num></code> Expression	Color Number or the <code><const></code> Expression	Color	Use of Color
OFF	COL_BACKGROUND	0	Depends on the GUI	Used for background
1	COL_HEADING	1	Gray-blue	Used for headers
2	COL_NORMAL	2	Light gray	Used for list bodies
3	COL_TOTAL	3	Yellow	Used for data total
4	COL_KEY	4	Blue-green	Used for key columns
5	COL_POSITIVE	5	Green	Used for a positive threshold value
6	COL_NEGATIVE	6	Red	Used for a negative threshold value
7	COL_GROUP	7	Violet	Used for control levels

The `INTENSIFIED` clause in the `FORMAT` statement is used to set the intensity of the background color. The default setting is `INTENSIFIED ON`. The use of the `COLOR OFF` setting changes the foreground color instead of the background color.

The `INVERSE` clause is used to determine whether or not the background or foreground color is set with the `COLOR` clause.

[Listing 6.9](#) shows various types of colors in a list:

Listing 6.9: Displaying various colors

```
REPORT ZCOLOR_DISPLAY_DEMO.
*-----*
*/ Data Declarations
*/
DATA i TYPE i VALUE 0.

DATA col(15) TYPE c.
*-----*
*/ While Loop
*/
WHILE i < 8.
CASE i.
WHEN 0.
    col = 'COL_BACKGROUND '.
WHEN 1.
    col = 'COL_HEADING '.
WHEN 2.
    col = 'COL_NORMAL '.
WHEN 3.
    col = 'COL_TOTAL '.
WHEN 4.
    col = 'COL_KEY '.
WHEN 5.
```



```
col = 'COL_POSITIVE ' .
WHEN 6.
col = 'COL_NEGATIVE ' .
WHEN 7.
col = 'COL_GROUP ' .
ENDCASE.
*-----*
*/FORMAT Statement
*/
FORMAT INTENSIFIED COLOR = i.
WRITE: /(4) i, AT 7 SY-VLINE,
col, SY-VLINE,
col INTENSIFIED OFF, SY-VLINE,
col INVERSE.
i = i + 1.
ENDWHILE.
*-----*
```

Listing 6.9 shows the use of the COLOR statement. Figure 6.15 shows the output of Listing 6.9:

0	COL_BACKGROUND	COL_BACKGROUND	COL_BACKGROUND
1	COL_HEADING	COL_HEADING	COL_HEADING
2	COL_NORMAL	COL_NORMAL	COL_NORMAL
3	COL_TOTAL	COL_TOTAL	COL_TOTAL
4	COL_KEY	COL_KEY	COL_KEY
5	COL_POSITIVE	COL_POSITIVE	COL_POSITIVE
6	COL_NEGATIVE	COL_NEGATIVE	COL_NEGATIVE
7	COL_GROUP	COL_GROUP	COL_GROUP

© SAP AG. All rights reserved.

Figure 6.15: Output of the COLORS clause in the list

In Figure 6.15, observe the different possibilities of colors that can be assigned to a list by using the INTENSIFIED and INVERSE clauses of the FORMAT statement. The INTENSIFIED clause is used to set the intensity of the background color.

Enabling Fields for INPUT

You can enable output fields in a list as input-enabled fields (accepting input data from users) using the FORMAT statement. The value of these fields can be modified by the user. You can print out the changes that you have made in interactive lists or process them later by using the READ LINE statement. Note that the variables that accept input data from users do not change. The following syntax of the FORMAT statement is used to create input-enabled fields:

```
FORMAT INPUT [ON|OFF].
```

The syntax to make the output fields input-enabled at runtime is:

```
FORMAT INPUT = i.
```

The ON clause (or i unequal to zero) is used to format the subsequent output as input-enabled fields. The background and foreground colors of the input-enabled fields are different as compared to the remaining list. The foreground color of an input field is changed by the INTENSIFIED clause. You can also make horizontal lines input-enabled by formatting them as input fields. Listing 6.10 shows the use of the FORMAT statement to specify the various formatting options:

Listing 6.10: Using the FORMAT statement to specify various formatting options

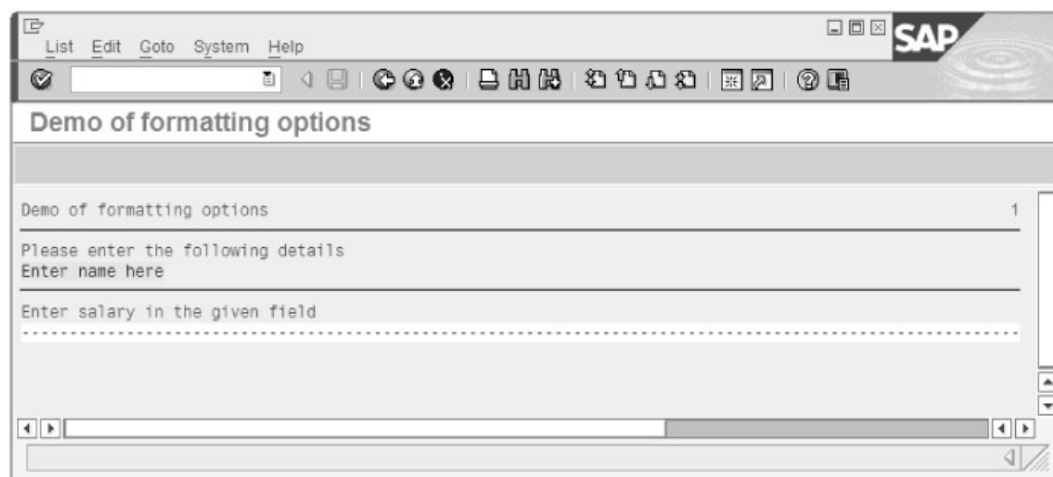
```
REPORT ZFORMATTING_DEMO.
WRITE 'Please enter the following details'.
WRITE /'Enter name here' INPUT ON.
ULINE.
WRITE 'Enter salary in the given field'.
*/Using FORMAT INPUT ON statement to make the output fields
input-enabled
```

```

FORMAT INPUT ON INTENSIFIED OFF.
ULINE.

```

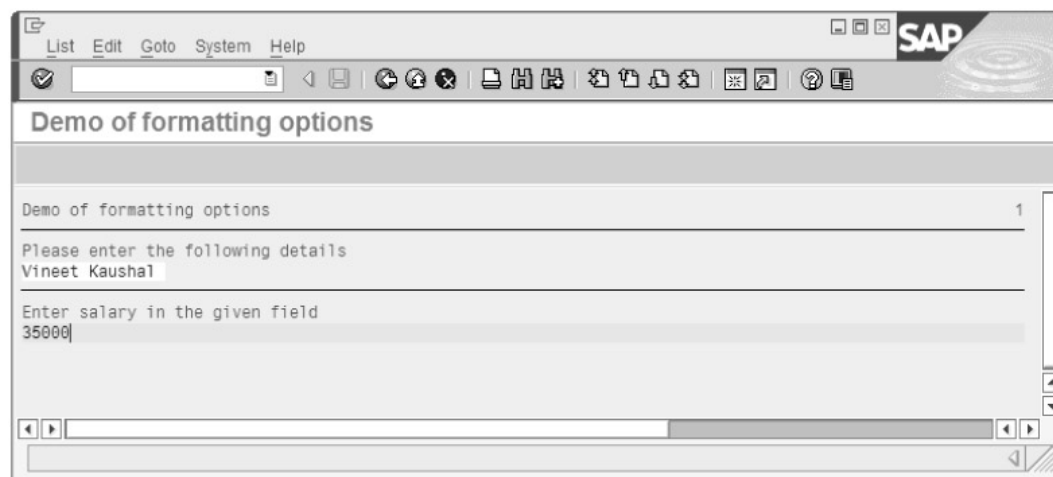
In this listing, the `INPUTON` clause is used with the `WRITE` statement to create an input field that accepts data from users. The `FORMAT` statement is used with the `INTENSIFIED OFF` clause so that the foreground color of the first input field is different from the other one. [Figure 6.16](#) shows the output of [Listing 6.10](#):



© SAP AG. All rights reserved.

Figure 6.16: Input fields in lists

In [Figure 6.16](#), note that the text "Enter name here" is an input field displayed in a different color. You can enter the value in the input field by deleting the existing characters in the fields. For instance, we have entered a name, Vineet Kaushal, in the "Enter name here" field and 35000 in place of the dotted lines (- - - -) field, as shown in [Figure 6.17](#):



© SAP AG. All rights reserved.

Figure 6.17: Input fields

Displaying Fields as Hotspots

The `FORMAT` statement can also be used to specify a field as a hotspot field, a special area on the output screen that triggers an event when a user clicks the field. The following syntax of the `FORMAT` statement is used to create a hotspot field:

```

FORMAT HOTSPOT [ON|OFF].

```

If you want to designate existing fields as hotspots, the following option of the `FORMAT` statement must be used:

```

FORMAT HOTSPOT = h.

```

The `ON` clause in the `FORMAT` statement is used to declare a field as a hotspot field. You cannot use the `HOTSPOT` clause

if `INPUT ON` (as discussed in the "[Enabling Fields for INPUT](#)" section) is set because the cursor cannot be positioned on an input field by using `HOTSPOT ON`. In addition, you cannot format the horizontal lines created with the `ULINE` clause and blank lines created with the `SKIP` clause as hotspots.

Exploring System Variables

System variables store the information relation to the SAP R/3 system in an ABAP Dictionary structure named `SYST`. The SAP R/3 system makes system variables available within your program. These variables are available to existing ABAP programs, and are updated automatically by the SAP system if the program's environment changes. All the SAP system variables are prefixed with the string `SY`. [Table 6.19](#) describes a list of some important system variables used in the SAP system:

Table 6.19: Describing system variables

System Variable	Description
<code>SY-INDEX</code>	Denotes the number of a loop pass.
<code>SY-PAGENO</code>	Denotes the current page number.
<code>SY-TABIX</code>	Denotes the current line index of an internal table
<code>SY-DBCNT</code>	Denotes the total number of processed lines of an internal table.
<code>SY-LSIND</code>	Denotes a list index.
<code>SY-MANDT</code>	Denotes the client number from logon.
<code>SY-LANGU</code>	Denotes the current language.

Dynamic Assignment

Data objects are accessed dynamically in ABAP programs by using field symbols and data references. Unlike static access to a data object, where you need to specify the name of the data object, you can represent dynamic data objects by using field symbols.

Field symbols are symbolic names for fields, which do not reserve space for a field physically but act as pointers to other fields. Any data object can be pointed to by a field symbol. The data object referenced by a field symbol is assigned after it has been declared in a program.

Before using a field symbol in a program, you must assign a field to the field symbol. This implies that whenever a field symbol is addressed in a program, you are actually addressing the field that is assigned to the field symbol.

Note Field symbols are similar to the dereferenced pointers in the C language (that is, pointers to which the content operator `*` is applied).

The operations performed on field symbols are actually applied to the fields assigned to them. For example, a `MOVE` statement assigned between two field symbols assigns the content of the source field symbol to the target field symbol.

Field symbols can be created either with or without type specifications. If the type is not assigned to a field symbol, it inherits all the technical attributes of the field assigned to it. If the type is specified, the SAP system checks whether a match exists between the assigned field and the field symbol. The following syntax is used to declare a field symbol:

```
FIELD-SYMBOLS <f> [typing].
```

The angular brackets (shown as `< >`) are used to identify the field symbols in a program.

When a data object is assigned to a field symbol, the field symbol inherits all the technical attributes of the data object. Therefore, the data type of the field symbol is actually the data type of the data object assigned to the field symbol.

Now, let's discuss the flow control statements of a processing block in an ABAP program.

Describing Flow Control Statements

Occasionally, ABAP programs use some conditions or loops to control the flow of execution of the ABAP program. Various conditions and loops can be implemented in programs by using the standard keywords, such as `IF`, `CASE`, `DO`, and `WHILE`.

As already discussed, the structure of an ABAP program is made up of processing blocks. In this section, you learn about controlling the flow of a program within a processing block. This is regarded as an internal control of an ABAP program as opposed to the external control provided by events in the ABAP runtime environment. According to the principles of structured programming, the internal flow of a processing block can be controlled by using control structures. The control structures divide the processing blocks into smaller statement blocks for easier processing of an ABAP program. **Figure 6.18** shows the flow control statements that a processing block may contain:

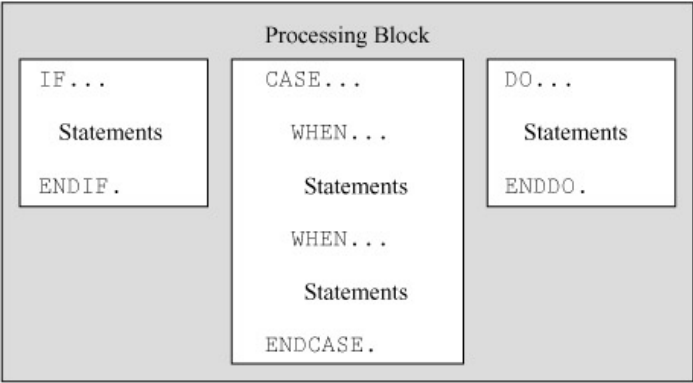


Figure 6.18: Flow control statements

The processing block of the ABAP program consists of several statements, some of which are:

- IF...ENDIF
- CASE...ENDCASE
- DO...ENDDO

Now, let's discuss each control statement in detail.

The IF...ENDIF Control Statement

IF...ENDIF is a control statement used to specify one or more conditions. You can also nest the IF control structures in an ABAP program. The following syntax is used for the IF...ENDIF statement:

```
IF <condition1>.  
    <statement block>.  
ELSEIF <condition2>  
    <statement block>.  
ELSEIF <condition3>  
    <statement block>.  
.....  
ELSE.  
    <statement block>.  
ENDIF.
```

In this syntax, the execution of the processing block is based on the result of one or more logical conditions associated with the processing block.

Table 6.20 shows the description of the clauses used in the IF...ENDIF control statement:

Table 6.20: Description of the clauses

Clauses	Descriptions
condition1	Represents a logical condition that evaluates a true or false condition.
condition2	Shows the second condition specified in the ELSEIF statement, which is executed when the IF statement condition turns out to be false.
ENDIF	Shows the end of the IF statement block.

The following guidelines must be kept in mind while constructing the `IF...ENDIF` control structure:

- Each `IF` statement must end with a matching `ENDIF` statement. The `ELSE` and `ELSEIF` statements are optional.
- You can use parentheses, but each parenthesis should be separated by a space. For example, `IF (a1 = a2) or (a3 = a4)` is correct, and `IF (s1 = s2) or (w1 = w2)` is incorrect.
- Variables can be compared with blanks and zeroes by using the `IS INITIAL` clause. For example, `IF f1 IS INITIAL` will be true if `f1` is of type `C` and blank. If `f1` is of any other data type, the statement is true if `f1` contains zeroes.
- To fulfill a negation condition, `NOT` must precede the logical expression, as shown in the following examples:
 - `IF NOT f1 IS INITIAL` is correct
 - `IF f1 IS NOT INITIAL` is incorrect
- Variables can be compared against null values by using the `IS NULL` clause. For example, `IF f1 IS NULL`.

Table 6.21 shows various types of comparison operators for different operands:

Table 6.21: Comparison operators

Comparison	Alternate Forms	True When
<code>a1 = a2</code>	<code>EQ</code>	<code>a1</code> equals <code>a2</code>
<code>a1 <> a2</code>	<code>NE</code> or <code>><</code>	<code>a1</code> does not equal <code>a2</code>
<code>a1 > a2</code>	<code>GT</code>	<code>a1</code> is greater than <code>a2</code>
<code>a1 < a2</code>	<code>LT</code>	<code>a1</code> is less than <code>a2</code>
<code>a1 >= a2</code>	<code>GE</code> or <code>=></code>	<code>a1</code> is greater than or equal to <code>a2</code>
<code>a1 <= a2</code>	<code>LE</code> or <code>= <</code>	<code>a1</code> is less than or equal to <code>a2</code>
<code>a1 between a2 and a3</code>	<code>N/A</code>	<code>a1</code> lies between <code>a2</code> and <code>a3</code> (inclusive)
<code>Not a1 between a2 and a3</code>	<code>N/A</code>	<code>a1</code> lies outside the range of <code>a2</code> to <code>a3</code> (inclusive)

In Table 6.21, `a1`, `a2`, and `a3` can be variables, literals, or field strings. Automatic conversion is performed in case of variables or literals, if the data type or length of the variables or the literals does not match. Similarly, automatic type adjustment is performed for either one or both of the values while comparing two values of different data types. The type of conversion is decided by the data type and the preference order of the data type. The order of preference of the data types is given as follows:

- If one field is of type `C`, the other type is automatically converted to the `F` type.
- If one field is of type `P`, the other is converted to the `P` type.
- If one field is of type `I`, the other is converted to the `I` type.
- If one field is of type `D`, the other is converted to the `D` type. However, the `C` and `N` types are not converted, but compared directly.
- If one field is of type `T`, the other is converted to the `T` type. However, the `C` and `N` types are not converted, but compared directly.
- If one field is of type `N` and other is of type `C` or `X`, both are converted to the type `P`.
- If one field is of the type `C` and the other is of type `X`, the `X` type is converted to the `C` type.

Note Field strings are `C` type variables.

Listing 6.11 shows an example of the `IF...ELSE...ENDIF` statement:

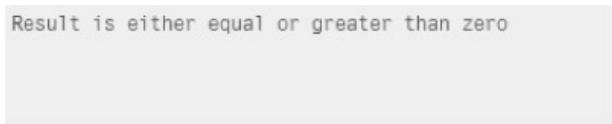
Listing 6.11: Using the `IF...ELSE...ENDIF` statement

```

REPORT ZIF_DEMO.
*-----*
*/ Data Declarations
*/
DATA RESULT TYPE I Value 45.
*-----*
*/IF Condition
IF RESULT < 0.
WRITE / 'Result is less than zero'.
ELSE.
WRITE / 'Result is either equal to or greater than zero'.
ENDIF.
*-----*

```

In [Listing 6.11](#), RESULT is a variable of type I, storing the value 45. The IF statement compares the value of the RESULT variable with zero. If the value of RESULT is less than zero, the output displayed is Result is less than zero; otherwise, the output is Result is either equal or greater than zero. [Figure 6.19](#) shows the output of [Listing 6.11](#):



Result is either equal or greater than zero

© SAP AG. All rights reserved.

Figure 6.19: Result of the IF statement

Using the ELSEIF Statement

The ELSEIF statement is used to avoid nesting of the IF statement, because the nesting of the IF statements can make the code difficult to understand. [Listing 6.12](#) shows how to use the ELSEIF statement:

Listing 6.12: Using the ELSEIF Statement

```

REPORT ZIF_DEMO.
*-----*
*/ Data Declarations
*/
DATA RESULT TYPE I Value 45.
*-----*
*/IF and ELSEIF Statements

IF RESULT < 0.
WRITE / 'Result is less than zero'.
ELSEIF RESULT < 50.
WRITE / 'Result is less than fifty'.
ELSE.
WRITE / 'Result is either equal to or greater than zero'.
ENDIF.
*-----*

```

In [Listing 6.12](#), RESULT is a variable of type I, storing the value 45. The IF statement checks whether the value of the RESULT variable is less than zero. If the value of RESULT is less than zero, the output displayed is Result is less than zero; if the value of RESULT is less than fifty, the output displayed is Result is less than fifty, else the output displayed is Result is either equal or greater than zero. [Figure 6.20](#) shows the output of [Listing 6.12](#):



Result is less than fifty

© SAP AG. All rights reserved.

Figure 6.20: Output of the ELSEIF statement

Working with Character String Operations

In ABAP, you can perform many operations on character strings. The operations, such as comparing the content and pattern of two character strings, can be performed easily with the help of some predefined operators. Table 6.22 shows noteworthy operators for string:

Table 6.22: Noteworthy string operators

Operator	Description	True When	Case-Sensitive	Trailing Blanks Ignored
a1 CO a2	a1 Contains Only a2	a1 is solely composed of the characters in a2	Yes	No
a1 CN a2	NOT a1 Contains Only a2	a1 contains characters that are not in a2	Yes	No
a1 CA a2	a1 Contains Any a2	a1 contains at least one character of a2	Yes	No
a1 NA a2	NOT a1 Contains Any a2	a1 does not contain any character of a2	Yes	No
a1 CS a2	a1 Contains a String a2	a1 contains the character string a2	No	Yes
a1 NS a2	NOT a1 Contains a String a2	a1 does not contain the character string a2	No	Yes
a1 CP a2	a1 Contains a Pattern a2	a1 contains the pattern in a2	No	Yes
a1 NP a2	NOT a1 Contains a Pattern a2	a1 does not contain the pattern in a2	No	Yes

Listing 6.13 shows how character string operations are performed using the IF...ELSEIF...ENDIF statement:

Listing 6.13: String operations using the IF...ELSEIF...ENDIF statement

```
REPORT demo_flow_control_if.
*-----*
*/ Data Declarations
*/
DATA: text1(30) TYPE c VALUE 'This is the first text',
      text2(30) TYPE c VALUE 'This is the second text',
      text3(30) TYPE c VALUE 'This is the third text',
      string(5) TYPE c VALUE 'eco'.
*-----*
*/Using IF..ELSEIF..ENDIF statement
IF text1 CS string.
WRITE / 'Condition 1 is fulfilled'.

ELSEIF text2 CS string.
WRITE / 'Condition 2 is fulfilled'.

ELSEIF text3 CS string.
WRITE / 'Condition 3 is fulfilled'.

ELSE.
WRITE / 'No condition is fulfilled'.
ENDIF.
*-----*
```

In Listing 6.13, the second logical expression (text2 CS string) is true because the string, eco, occurs in text2. Figure 6.21 shows the output of Listing 6.13:



Figure 6.21: Output of the IF..ELSEIF...ENDIF statement

The CASE Control Statement

The CASE control statement is used when you need to compare two or more numbers. The syntax of the CASE statement is:

```
CASE <f> .
WHEN <fij> [ OR <fij> OR.. ... ..].
    <statement block>
    WHEN <fij> [ OR <fij> OR.. ... ..].
    <statement block>
WHEN... ..
WHEN OTHERS.
    <statement block>
ENDCASE.
```

In the preceding syntax, the statement block following a WHEN clause is executed if the content of the fields shown in the <f> expression is similar to one of the fields <fij>. After executing all the conditions specified in the WHEN statement, the program continues to process the remaining statements after the ENDCASE statement. The WHEN OTHERS clause is executed in a program when the value of the <f> field does not match with any value specified in the <fij> fields of the WHEN clause.

Consider the following key points while constructing the CASE... WHEN... ENDCASE statement:

- If the WHEN OTHERS clause is omitted in a program and the value of the <f> field does not match with any value specified in the <fij> fields, the program continues to process the remaining statements after the ENDCASE statement.
- No logical expressions can be used for the <f> field.
- The field strings used in the CASE... WHEN... ENDCASE statement are treated as type C variables.

There is a difference between the CASE and IF... ELSE control statement. In the IF... ELSE statement, you can use the complex expression in the <conditions> clause, but in the CASE statement, you can only give a single value in the <f> field. [Listing 6.14](#) shows an example of the CASE statement:

Listing 6.14: Using the CASE statement

```
REPORT demo_flow_control_case.
*-----*
*/ DATA Declaration
*/
DATA: a1    TYPE c VALUE 'X',
a2    TYPE c VALUE 'Y',
a3    TYPE c VALUE 'Z',
string TYPE c VALUE 'A'.
*-----*
*/ CASE Control Statement
CASE string.
WHEN a1 OR a2.
WRITE: / 'String is', a1, 'OR', a2.
WHEN a3.
WRITE: / 'String is', a3.
WHEN OTHERS.
WRITE: / 'String is not', a1, a2, a3.
ENDCASE.
*-----*
```

[Listing 6.14](#) defines three variables (a1, a2, and a3) with their corresponding values. Note that none of the values of the string variables matches the WHEN condition. In such a situation, the statement block with WHEN OTHERS is executed and the corresponding values of the a1, a2, and a3 variables are displayed. [Figure 6.22](#) shows the output of [Listing 6.14](#):

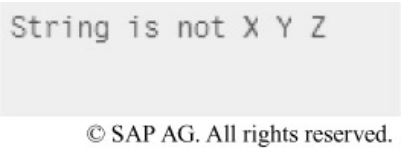


Figure 6.22: Output of the CASE statement

Looping

A statement block is executed repeatedly in a program by using loops in the program. The loops are categorized into two types:

- Unconditional loops by using the DO statement.
- Conditional loops by using the WHILE statement

Now, let's discuss each in detail.

Unconditional Loops Using the DO Statement

Unconditional loops repeatedly execute several statements without specifying any condition. The DO statement implements unconditional loops by executing a set of statements (statement block) several times unconditionally. The syntax of the DO statement is:

```
DO [n TIMES] [VARYING <f> FROM <f1> NEXT <f2>].  
<statement block>  
ENDDO.
```

Table 6.23 shows the clauses used in the syntax of the DO statement:

Table 6.23: Clauses in the DO statement

Clause	Description
TIMES	Imposes a restriction on the number of loop passes, represented by n. It is necessary that the value of n is not negative or 0. If it is so, the statements in the loop are not executed. If the TIMES clause is not used, at least any one of the two statements (EXIT or STOP) should be used to avoid endless loops.
VARYING	Shows that new values of the <f> variable can be assigned in each loop pass.

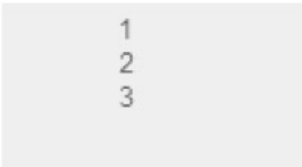
Note The DO Statement can be nested as well as combined with other loop forms.

Listing 6.15 shows an example of the DO statement:

Listing 6.15: Using the DO statement

```
REPORT demo_do.  
*-----*  
*/Using DO Statement  
*/  
DO.  
WRITE / SY-INDEX.  
IF SY-INDEX = 3.  
EXIT.  
ENDIF.  
ENDDO.  
*-----*
```

In Listing 6.15, the process passes through the loop three times and then leaves the DO loop when the value of the SY-INDEX system variable becomes 3. Figure 6.23 shows the output of Listing 6.15:



© SAP AG. All rights reserved.

Figure 6.23: Output of the DO statement

Listing 6.16 shows the nesting of the DO statement:

Listing 6.16: Nested DO statement

```
*-----*
*/ Nested DO Statement
DO 2 TIMES.
WRITE SY-INDEX.
SKIP.
    DO 3 TIMES.
WRITE / SY-INDEX.
ENDDO.
SKIP.
ENDDO.
*-----*
```

Listing 6.16 shows a nesting of the DO loop is processed twice. The inner DO loop is processed three times each time the outer DO loop is processed. That is, the inner DO loop is processed six times in this case. The SY-INDEX system variable is used to store the value of the number of loops passed. Figure 6.24 shows the output of Listing 6.16:



© SAP AG. All rights reserved.

Figure 6.24: Output of the nested DO loop

Conditional Loops Using the WHILE Statement

Conditional loops execute several statements only after certain conditions are fulfilled. This can be accomplished with the WHILE statement. The WHILE statement allows you to execute a block of statements so long as the condition mentioned in the WHILE statement evaluates to true. The syntax of the WHILE statement is:

```
WHILE <condition> [VARY <f> FROM <f1> NEXT <f2>]
[statement_block]
ENDWHILE.
```

Table 6.24 describes the clauses used in the syntax of the WHILE statement:

Table 6.24: Clauses used in the WHILE statement

Clause	Description
<condition>	Represents a logical expression.

[statement block]	Specifies a set of statements executed when the condition specified in the WHILE statement is true.
[VARY <f> FROM <f1> NEXT <f2>]	Specifies the condition to be verified, as specified in the <f> field. The range of values for <f> is specified from the <f1> field to the <f2> field.

The block of statements between the WHILE and ENDWHILE control statements is executed so long as the condition is evaluated to true or until a termination statement, such as EXIT or STOP, is reached. The SY-INDEX system variable contains the number of loop passes, including the current loop pass.

You can nest the WHILE loop any number of times and combine the nested WHILE loop with other loop forms. Listing 6.17 shows an example of the WHILE statement:

Listing 6.17: Using the WHILE statement

```
REPORT ZWHILE_DEMO.
*-----*
*/ Data Declarations
*/
DATA: LENGTH TYPE I VALUE 0,
VAR1 TYPE I VALUE 0,
TEXT1(50) TYPE C VALUE 'Calculating the Length of the Text
String'.
*-----*
VAR1 = STRLEN(TEXT1).
*-----*
*/Using WHILE statement
WHILE TEXT1 NE SPACE.
WRITE TEXT1(1).
LENGTH = SY-INDEX.
SHIFT TEXT1.
ENDWHILE.
*-----*
WRITE: / 'STRLEN: ', VAR1.
WRITE: / 'Length of string:', LENGTH.
```

Listing 6.17 shows a WHILE loop used to determine the length of a character string. This is done by shifting the string one position to the left each time the loop is processed. This loop is processed until it contains only blanks. Figure 6.25 displays the result of Listing 6.17:



© SAP AG. All rights reserved.

Figure 6.25: Displaying the result of the WHILE loop

Terminating Loops

A loop can be terminated prematurely with the help of two types of termination statements provided by ABAP. The two types of termination statements are:

- Statements that apply to a loop, such as CONTINUE, CHECK, and EXIT.
- Statements that apply to an entire processing block, such as STOP and REJECT.

The CONTINUE statement can be used only in a loop statement. On the other hand, the CHECK and EXIT statements are context-sensitive. When used in the loop, the CHECK and EXIT statements are responsible for the execution of the loop itself. However, outside the loop, the statements terminate the entire processing block, such as a subroutine, dialog module, or event block, in which they occur. CONTINUE, CHECK, and EXIT can be used in looping statements, such as DO, WHILE, LOOP, and SELECT.

A loop can be terminated conditionally or unconditionally. Now, let's learn about each of the two possible cases in detail.


Terminating a Loop Pass Unconditionally

The `CONTINUE` statement is used in a statement block of the loop to terminate a single loop pass immediately and unconditionally. As soon as the `CONTINUE` statement is executed, the execution of the remaining statements in the current processing block is stopped and the next loop pass is started. [Listing 6.18](#) shows the working of the `CONTINUE` statement:

Listing 6.18: Using the `CONTINUE` statement

```
REPORT demo_continue.
DO 4 TIMES.
IF SY-INDEX = 2.
*/Using CONTINUE statement after which the statements are not
evaluated
CONTINUE.
ENDIF.
WRITE / SY-INDEX.
ENDDO.
```

[Listing 6.18](#) shows that a loop is executed four times but a condition is inserted with the value of the `SY-INDEX` system variable equal to 2. The specified condition is true for the second loop pass, so the `WRITE` statement is not executed. The `CONTINUE` statement ignores the statements in the current statement block and continues with the next loop pass. [Figure 6.26](#) shows the output of [Listing 6.18](#):



© SAP AG. All rights reserved.

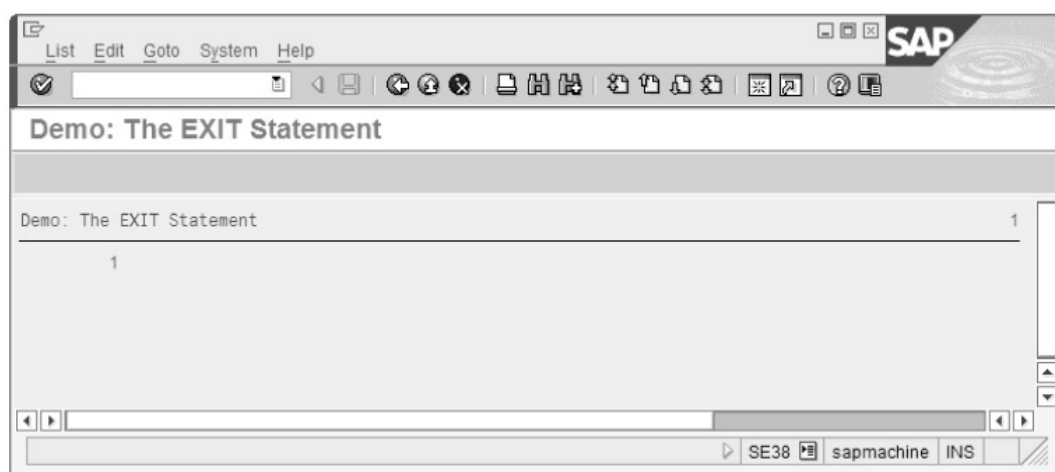
Figure 6.26: Output of the `CONTINUE` statement

Apart from the `CONTINUE` statement, the `EXIT` statement is used to terminate an entire loop immediately and unconditionally. As soon as the `EXIT` statement is executed, the loop is terminated and the statements following the structure of the loop are processed. If the `EXIT` statement is used in a nested loop, only the current loop is executed after the `EXIT` statement is executed. [Listing 6.19](#) shows the working of the `EXIT` statement:

Listing 6.19: Using the `EXIT` statement

```
REPORT demo_exit.
*-----*
*/ DO Loop
*/
DO 3 TIMES.
IF SY-INDEX = 2.
EXIT.                "EXIT Statement
ENDIF.
WRITE SY-INDEX.
ENDDO.
*-----*
```

In [Listing 6.19](#), the `DO` loop is terminated in the second loop pass and the `WRITE` statement is not executed. [Figure 6.27](#) shows the output of [Listing 6.19](#):



© SAP AG. All rights reserved.

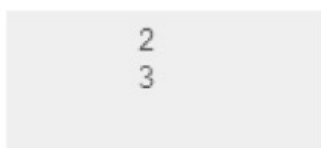
Figure 6.27: Output of the EXIT statement**Terminating a Loop Pass Conditionally**

The `CHECK` statement terminates a loop pass based on a condition. If the condition specified in the `CHECK` statement is evaluated to false, the remaining statements in the statement block, after the `CHECK` statement, are ignored and the next loop pass starts. The condition specified in the `CHECK` statement can be any logical expression. Listing 6.20 shows the working of the `CHECK` statement:

Listing 6.20: Using the CHECK statement

```
REPORT demo_check.
*-----*
*/DO statement using the CHECK statement
*/
DO 4 TIMES.
CHECK SY-INDEX BETWEEN 2 and 3.
WRITE / SY-INDEX.
ENDDO.
*-----*
```

In Listing 6.20, you see that the first and fourth loop passes are terminated without the execution of the `WRITE` statement because the value of the `SY-INDEX` system variable does not lie between 2 and 3. Figure 6.28 shows the output of Listing 6.20:



© SAP AG. All rights reserved.

Figure 6.28: Output of the CHECK statement**Summary**

In this chapter, you have learned about the ABAP program structure, its various components, and ABAP Editor, which is an ABAP Workbench tool used to develop ABAP programs. Next, you have learned about the programs created in ABAP Editor. You have also learned to use the various options present on the initial screen of ABAP Editor and add comments to a program. In addition, you have learned about types and objects in an ABAP program. This chapter also discusses the various statements used to define data in ABAP programs. Further, you have learned about various kinds of assignment statements, such as `MOVE`, `MOVE-CORRESPONDING`, `WRITE TO`, and `CLEAR`. Next, you have learned about the different formatting options available in ABAP, such as the `FORMAT` and `WRITE` statements. Moreover, you have learned about the flow control statements, such as the `IF` and `CASE` statements. Finally, you have learned to work with loops.

