Chapter 4
ABAP Programming Concepts

In this chapter, we'll look at basic ABAP programming language concepts, which will lay the foundations to write your own ABAP programs.

ABAP programs process data from the database or an external source. As discussed in Chapter 2, ABAP programs run in the application layer, and ABAP program statements can work only with locally available data in the program. External data, such as user inputs on screens (presentation layer), data from a sequential file (presentation layer), or data from a database table (database layer), must always be transported to and saved in the program's local memory (application layer) to be processed with ABAP statements.

In other words, before any data can be processed with an ABAP program, it must first be read from the source, stored locally in the program memory, and then accessed via ABAP statements; you can't work with the external data directly using ABAP statements. This temporary data that exists in an ABAP program while the program is executed is called *transient data* and is cleared from memory after the program execution ends. If you want to access this transient data later, it must be stored persistently in the database; such stored data is called *persistent data*.

This chapter provides information on the basic building blocks of the ABAP programming language, which helps explain the different ways to store and process data locally in ABAP programs. In Section 4.1, we begin by looking at the general structure of an ABAP program, before diving into ABAP syntax rules and ABAP keywords in Section 4.2 and Section 4.3, respectively.

Section 4.4 introduces the TYPE concept and explores key elements, such as data types and data objects, which define memory locations in ABAP programs to store data locally. In Section 4.5, we discuss the different types of ABAP statements that can be employed, including declarative, modularization, control, call, and operational statements. Finally, in Section 4.6, we conclude this chapter by walking through the steps to create your first ABAP program.

Initially, we'll use a procedural model for our examples to keep things simple; we'll switch to an object-oriented model after we discuss object-oriented programming (OOP) in Chapter 8.

4.1 General Program Structure

Every ABAP program consists of a global declaration area and a procedural area. The global declaration area is typically used to define memory locations (called data objects) to store the data that can then be processed by ABAP statements typically written in the procedural area of the program. Therefore, any ABAP program can be broadly divided into two parts:

- **Global declarations**
  Global data for the program is defined that can then be accessed from anywhere in the program.
- **Procedural**
  The procedural part of the program consists of various processing blocks, such as dialog modules, event blocks, and procedures. The statements within these processing blocks can access the global data defined in the global declarations.

In this section, we'll look at both of these program structure parts.

4.1.1 Global Declarations

The global declaration part of an ABAP program uses declarative statements to define memory locations, which are called *data objects* and store the work data of the program. ABAP statements work only with data available in the program as content for data objects, so it's imperative that data is stored in a data object before it's processed by an ABAP program.

Data objects are local to the program; they can be accessed via ABAP statements in the same program. The global declaration area exists at the top of the program (see Figure 4.1). We use the term *global* with respect to the program; data objects defined in this area are visible and valid throughout the entire ABAP program and can be accessed from anywhere in the source code using ABAP statements. Here, you can see the:

❶ Declaration part

❷ Procedural part

ABAP also uses *local declarations*, which can be accessed only within a subset of the program. We'll explore local declarations in Chapter 7 when we discuss modularization techniques.

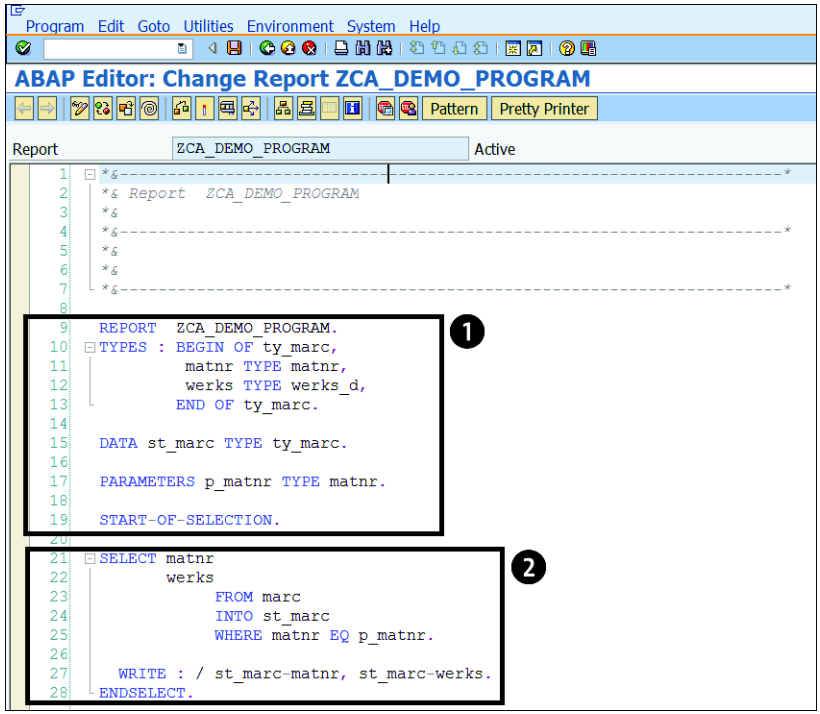


Figure 4.1 Program Structure

4.1.2 Procedural Area

After the declarative area is the procedural portion of the program. Here, the processing logic of the ABAP program is defined. In this section of the ABAP program, you typically use various ABAP statements to import and process data from an external source.

The procedural area is where the program logic is implemented. It uses various processing blocks that contain ABAP statements to implement business requirements. For example, in a typical report, the procedural area contains an event block to process the selection screen (user input screen) and to validate user inputs on the selection screen, uses another event block to fetch the data from the database based on user input, and then calls a procedure to display the output to the user.

4.2 ABAP Syntax

The source code of an ABAP program is simply a collection of various ABAP statements that are interpreted by the runtime environment to perform specific tasks. You use declarative statements to define data objects, modularization statements to define processing blocks, and database statements to work with the data in the database.

In this section, we'll look at the basic syntax rules that every ABAP programmer should know. We'll then look at the use of chained statements and comment lines.

4.2.1 Basic Syntax Rules

There are certain basic syntax rules that need to be followed while writing ABAP statements:

- An ABAP program is a collection of individual ABAP statements that exist within the program. Each ABAP statement is concluded with a period (.), and the first word of the statement is known as a keyword.
- An ABAP statement consists of operands, operators, or additions to keywords (see Figure 4.2). The first word of an ABAP statement is an ABAP keyword; the remaining can be operands, operators, or additions. *Operands* are the data objects, data types, procedures, and so on.

  Various operators are available, such as assignment operators that associate the source and target fields of an assignment (e.g., `=` or `?=`), arithmetic operators that assign two or more numeric operands with an arithmetic expression (e.g., `+`, `-`, `*`), relational operators that associate two operands with a logical expression (e.g., `=`, `<`, `>`), and so on. Each ABAP keyword will have its own set of additions.
- Each word in the statement must be separated by at least one space.
- An ABAP statement ends with a period, and you can write a new statement on the same line or on a new line. A single ABAP statement can be extended over several lines.
- ABAP code isn't case-sensitive.

In Figure 4.2, the program shown consists of three ABAP statements written across three lines. The first word in each of these statements (`REPORT`, `PARAMETERS`, and `WRITE`) is a keyword. As you can see, each statement begins with a keyword and ends with a period. These contain a:

❶ Keyword

❷ Operand

❸ Addition

In addition, each ABAP word is separated by a space.

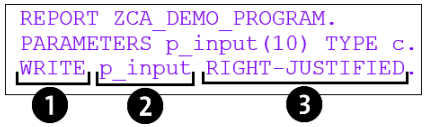


Figure 4.2 ABAP Statement

You can write multiple statements on one line or one statement can extend over multiple lines. Therefore, if you want, you can rewrite the code in Figure 4.2 as shown:

```
REPORT ZCA_DEMO_PROGRAM. PARAMETERS p_input(10) TYPE c. WRITE p_input RIGHT-
JUSTIFIED.
```

However, to keep the code legible, we recommend restricting your program to one statement per line. In some cases, it's recommended to break a single statement across multiple lines, for example:

```
SELECT * FROM mara INTO TABLE it_mara WHERE matnr EQ p_matnr.
```

The preceding statement may be written as shown in Listing 4.1 to make it more legible.

```
SELECT * FROM mara
         INTO TABLE it_mara
         WHERE matnr EQ p_matnr.
```

Listing 4.1 Splitting a Statement across Multiple Lines

4.2.2 Chained Statements

If more than one statement starts with the same *keyword*, you can use a colon (:) as a chain operator and separate each statement with a comma. These *chained statements* help you avoid repeating the same keyword on each line.

For example:

```
DATA v_name(20) TYPE c.
DATA v_age TYPE i.
```

can also be written as:

```
DATA : v_name(20) TYPE c,
       v_age TYPE i.
```

End the last statement in the chain with a period. Chained statements aren't limited to keywords; you can put any identical first part of a chain of statements before the colon and write the remaining parts of the individual statements separated by a comma, for example:

```
v_total = v_total + 1.
v_total = v_total + 2.
v_total = v_total + 3.
v_total = v_total + 4.
```

can be chained as:

```
v_total = v_total + : 1, 2, 3, 4.
```

Note

ABAP code isn't case-sensitive, so you can use either uppercase or lowercase to write ABAP statements. We recommend writing keywords and their additions in uppercase and using lowercase for other words in the statement to make the code more legible.

4.2.3 Comment Lines

To make your source code easy to understand for other programmers, you can add comments to it (see Listing 4.2). *Comment lines* are ignored by the system when the program is generated, and they're useful in many ways.

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE n LENGTH 2.
*This is a comment line
f2 = f1.
WRITE f2. "This is also a comment line
```

Listing 4.2 Comment Lines

There are two ways to add comment lines in source code:

- You can enter an asterisk (*) at the beginning of a line to make the entire line a comment.
- You can enter a double quotation mark (") midline to make the part of the line after the quotation mark a comment (this is called an *in-line comment*).

You can *comment* (i.e., set as a comment) on a block of lines at once (a *multiline comment*) by selecting the lines to be commented on and pressing Ctrl+< on the keyboard. Similarly, to *uncomment* (i.e., set as normal code) a block of lines, you can select the lines and press Ctrl+>.

Alternatively, you can also use the context menu to comment or uncomment code. To comment a line of code or a block lines, select the code, right-click, and select the appropriate option from the **Format** context menu. This helps you avoid the tedious job of adding asterisks manually at the beginning of each line.

Now that you have a better understanding of basic ABAP syntax rules and chaining ABAP statements, in the next section, we'll look at the keywords used in ABAP.

4.3 ABAP Keywords

Because each ABAP statement starts with a keyword, writing ABAP statements is all about choosing the right keyword to perform the required task. Every keyword provides specific functionality and comes with its own set of *additions* that allow you to extend the keyword's functionality.

For each keyword, SAP maintains extensive documentation, which serves as a guide to understanding the syntax to use with the keyword and the set of additions supported for the keyword.

You can access the keyword documentation by typing "ABAPDOCU" in the command bar to open it just like any other transaction or by placing the cursor on the keyword and pressing F1 while writing your code in the ABAP Editor. You can also visit *https://s-prs.co/474905*, to access an online version of the ABAP keyword documentation (see Figure 4.3).

Because there are literally hundreds of keywords that can be used in ABAP, the best way to become familiar with the keywords is to explore them in relation to a requirement. We'll be taking this approach throughout the book as we introduce you to these keywords in various examples.

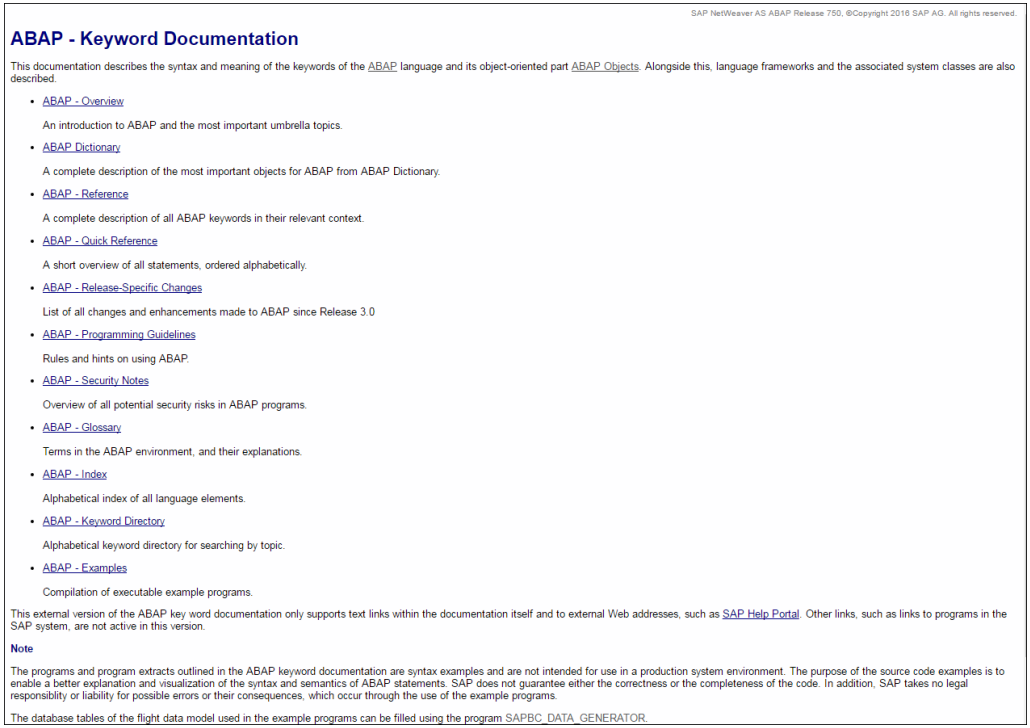


Figure 4.3 ABAP Keyword Documentation

4.4 Introduction to the TYPE Concept

An ABAP program only works with data inside data objects. The first thing you do when developing a program is declare data objects. Inside these data objects, you store the data to be processed in your ABAP program. You use declarative statements called *data declarations* to define data objects that store data in the program.

Typically, you'll work with various kinds of data, such as a customer's name, phone number, or amount payable. Each type of data has specific characteristics; for example, the customer's name consists of letters, a phone number consists of digits from 0 to 9, and due amount to the customer will be a number with decimal values. Identifying the type and length of data that you plan to store in data objects is what the TYPE concept is all about.

In this section, we'll look at data types, domains, and data objects.

4.4.1 Data Types

Data types are templates that define data objects. A data type determines how the contents of a data object are interpreted by ABAP statements. Other than occupying some space to store administrative information, they don't occupy any memory space for work data in the program. Their purpose simply is to supply the technical attributes of a data object.

Data types can be broadly classified as elementary, complex, or reference types. In this chapter, we'll primarily explore the elementary data types and will only briefly cover complex types and reference types. More information on complex data types can be found in Chapter 5, and more details about reference types are provided in Chapter 8.

Elementary Data Types

Elementary data types specify the types of individual fields in an ABAP program. Elementary data types can be classified as predefined elementary types or user-defined elementary types. We'll look at these subtypes next.

Predefined Elementary Data Types

The SAP system comes built in with predefined elementary data types. These data types are predefined in the SAP NetWeaver AS ABAP kernel and are visible in all ABAP programs. You can use these predefined elementary data types to assign a *type* to your program data objects. You can also create your own data types (user-defined elementary data types) by referring to the predefined data types.

Table 4.1 lists the available predefined elementary data types.

Data Type	Definition
`i`	4-byte integer
`int8`	8-byte integer
`f`	Binary floating-point number
`p`	Packed number
`decfloat16`	Decimal floating-point number with 16 decimal places
`decfloat34`	Decimal floating-point number with 34 decimal places

Table 4.1 Predefined Elementary Data Types

Data Type	Definition
c	Text field (alphanumeric characters)
d	Date field (*format:* YYYYMMDD)
n	Numeric text field (numeric characters 0 to 9)
t	Time field (*format:* HHMMSS)
x	Hexadecimal field

Table 4.1 Predefined Elementary Data Types (Cont.)

Predefined elementary data types can be classified as numeric or nonnumeric types. There are six predefined numeric elementary data types:

- 4-byte integer (`i`)
- 8-byte integer (`int8`)
- Binary floating-point number (`f`)
- Packed number (`p`)
- Decimal floating point number with 16 decimal places (`decfloat16`)
- Decimal floating point number with 34 decimal places (`decfloat34`)

There are five predefined nonnumeric elementary data types:

- Text field (`c`)
- Numeric character string (`n`)
- Date (`d`)
- Time (`t`)
- Hexadecimal (`x`)

The field length for data types `f`, `i`, `int8`, `decfloat16`, `decfloat34`, `d`, and `t` is fixed. In other words, you don't need to specify the length when you use these data types to declare data objects (or user-defined elementary data types) in your program. The field length determines the number of bytes that the data object occupies in memory. In types `c`, `n`, `x`, and `p`, the length isn't part of the type definition. Instead, you define it when you declare the data object in your program.

Before we discuss the predefined elementary data types further, let's see how they're used to define a data object in the program.

The keyword used to define a data object is DATA. For the syntax, you provide a name for your data object and use the addition TYPE to refer it to a data type, from which the data object can derive its technical attributes. For example, the line of code in Figure 4.4 defines the data object V_NAME of TYPE c (character). Here, you can see the:

❶ Keyword

❷ Data object name

❸ Data type reference

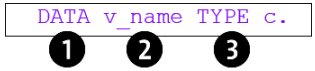


Figure 4.4 Declaring a Data Object

Notice that we didn't specify the length in Figure 4.4; therefore, the object will take the data type's initial length by default. Here, V_NAME will be a data object of TYPE c (text field) and LENGTH 1 (default length). It can store only one alphanumeric character.

If you want a different length for your data object, you need to specify the length while declaring the data object, either with parentheses or using the addition LENGTH:

```
DATA v_name(10) TYPE c.
DATA v_name TYPE c LENGTH 10.
```

In this example, V_NAME will be a data object of TYPE c and LENGTH 10. It can now store up to 10 alphanumeric characters. The Valid Field Length column in Table 4.2 lists the maximum length that can be assigned to each data type.

For data types of fixed lengths, you don't need to specify the length because it's part of the TYPE definition, for example:

```
DATA count TYPE i.
DATA date TYPE d.
```

Table 4.2 lists the initial length of each data type, its valid length, and its initial value. The initial length is the default length that the data object occupies in memory if no length specification is provided while defining the data object, and the initial value of a data object is the value it stores when the memory is empty.

For example, as shown in Table 4.2, a TYPE c data object will be filled with spaces initially, whereas a TYPE n data object would be filled with zeros.

Data Type	Initial Field Length (Bytes)	Valid Field Length (Bytes)	Initial Value
Numeric Types			
i	4	4	0
int8	8	8	0
f	8	8	0
p	8	1–16	0
decfloat16	8	8	0
decfloat34	16	16	0
Character Types			
c	1	1–65535	Space
d	8	8	'00000000'
n	1	1–65535	'0 … 0'
t	6	6	'000000'
Hexadecimal Type			
x	1	1–65535	X'0 … 0'

Table 4.2 Elementary Data Types: Technical Specifications

Predefined nonnumeric data types can be further classified as follows:

- **Character types**
  Data types `c`, `n`, `d`, and `t` are character types. Data objects of these types are known as *character fields*. Each position in one of these fields can store one code character. For example, a data object of `LENGTH 5` can store 5 characters, and a data object of `LENGTH 10` can store 10 characters. Currently, ABAP only works with single-byte codes, such as American Standard Code for Information Interchange (ASCII) and Extended Binary Coded Decimal Interchange Code (EBCDI):
  - ASCII is a character-encoding standard. ASCII code represents text in computers and other devices and is a popular choice for many modern character-encoding schemes. In an ASCII file, each alphabetic or numeric character is represented with a 7-bit binary number.

  - EBCDI is an 8-bit character encoding standard, which means each alphabetic or numeric character is represented with an 8-bit binary number.

  As of release 6.10, SAP NetWeaver AS ABAP supports both Unicode and non-Unicode systems. However, support for non-Unicode systems has been withdrawn from SAP NetWeaver 7.5.

  Single-byte code refers to character encodings that use exactly 1 byte for each character. For example, the letter A will occupy 1 byte in single-byte encoding.

- **Hexadecimal types**
  The data type `x` interprets individual bytes in memory. These fields are called *hexadecimal fields*. You can process single bits using hexadecimal fields. Hexadecimal notation is a human-friendly representation of binary-coded values. Each hexadecimal digit represents 4 binary digits (bits), so a byte (8 bits) can be more easily represented by a two-digit hexadecimal value.

With modern programming languages, we seldom need to work with data at the bits and bytes levels. However, unlike ABAP (which uses single-byte code pages, e.g., ASCII), many external data sources use multibyte encodings. Therefore, `TYPE x` fields are more useful in a Unicode system, in which you can work with the binary data to be processed by an external software application.

For example, you can insert the hexadecimal code 09 between fields to create a tab-delimited file so that spreadsheet software—such as Microsoft Excel—knows where each new field starts. `TYPE x` fields are also useful if you want to create files in various formats from your internal table data.

Predefined Elementary ABAP Types with Variable Lengths

The data types we've discussed so far either have a fixed length or require a length specification as part of data object declaration. We assume that we know the length of the data we plan to process; for example, a material number is defined in SAP as an alphanumeric field with a maximum of 18 characters in length. Therefore, if you want to process a material number in your ABAP program, you can safely create a data object as a `TYPE c` field with `LENGTH 18`. However, there can be situations in which you need a field with dynamic length because you won't know the length of the data until runtime. For such situations, ABAP provides data types with variable lengths:

- **`string`**
  This character type with a variable length can contain any number of alphanumeric characters.

- **xstring**
  This is a hexadecimal type with a variable length.

When you define a `string` as a data object, only the string header, which holds administrative information, is created statically. The initial length of the `string` data object is 0, and its length changes dynamically at runtime based on the data stored in the data object, for example:

```
DATA path TYPE string.
DATA xpath TYPE xstring.
```

Table 4.3 highlights some of the use cases for the predefined elementary data types.

Data Type	Use Case
Numeric Types	
`i`	Use to process integers such as counters, indexes, time periods, and so on. Valid value range is -2147483648 to +2147483647. If the value range of `i` is too small, use `TYPE p` without the `DECIMALS` addition. Example syntax: `DATA f1 TYPE i. "fixed length`
`f`	Use to process large values when rounding errors aren't critical. To a great extent, `TYPE f` is replaced by `decfloat` (`decfloat16` and `decfloat34`) as of SAP NetWeaver AS ABAP 7.1. Use data type `f` only if performance-critical algorithms are involved and accuracy isn't important. Example syntax: `DATA f1 TYPE f. "fixed length`

`p`	Use type `p` when fractions are expected with fixed decimals known at design time (distances, amount of money or quantities, etc.).
	Example syntax:
	`DATA f1 TYPE p DECIMALS 2. "Takes default length 8`
	`* OR you can manually specify length.`
	`DATA f1(4) TYPE p DECIMALS 3.`

Table 4.3 Predefined Elementary Data Types and Use Cases

Data Type	Use Case
decfloat16 and decfloat34	If you need fractions with a variable number of decimal places or a larger value range, use decfloat16 or decfloat34. Example syntax: DATA f1 TYPE decfloat16. DATA f1 TYPE decfloat34.
Nonnumeric Types	
c	Use to process alphanumeric values such as names, places, or any character strings. Example syntax: DATA f1 TYPE c. DATA f1(10) TYPE c. DATA f1 TYPE c LENGTH 10.
n	Use to process numeric values such as phone numbers or zip codes. Example syntax: DATA f1 TYPE n. DATA f1(10) TYPE n. DATA f1 TYPE n LENGTH 10.
d	Use to process dates with the expected format of YYYYMMDD. Example syntax: DATA f1 TYPE d.
t	Use to process time with the expected format of HHMMSS. Example syntax: DATA f1 TYPE t.
x	Use to process the binary value of the data. This is useful for working with different code pages. Example syntax: DATA f1 TYPE x. DATA f1(10) TYPE x. DATA f1 TYPE x LENGTH 10.

Table 4.3 Predefined Elementary Data Types and Use Cases (Cont.)

Data Type	Use Case
Nonnumeric Types	
`string`	Use when the length of the `TYPE c` field is known only at runtime. Example syntax: `DATA f1 TYPE string.`
`xstring`	Use when the length of the `TYPE x` field is known only at runtime. Example syntax: `DATA f1 TYPE xstring.`

Table 4.3 Predefined Elementary Data Types and Use Cases (Cont.)

For arithmetic operations, use numeric fields only. If nonnumeric fields are used in arithmetic operations, the system tries to convert the fields to numeric types automatically before applying the arithmetic operation. This is called *type conversion*, and each data type has specific *conversion rules*. Let's look at this concept in more detail next.

Type Conversions

When you move data between data objects, either the data objects involved in the assignment should be similar (both data objects should be of the same type and length), or the data type of the source field should be convertible to the target field.

If you move data between dissimilar data objects, then the system performs type conversion automatically by converting the data in the source field to the target field using the conversion rules. For the conversion to happen, a conversion rule should exist between the data types involved.

For example, the code in Listing 4.3 assigns a character field to an integer field.

```
DATA f1 TYPE c LENGTH 2 VALUE 23.
DATA f2 TYPE i.
f2 = f1.
```

Listing 4.3 Type Conversion with Valid Content

In Listing 4.3, the data object `f1` is defined as a character field with an initial value of `23`. Because `f1` is of `TYPE c`, the value is interpreted as a character by the system rather than an integer. The listing also defines another data object, `f2`, as an integer type.

When you assign the value of `f1` to the data object `f2`, the system performs the conversion using the applicable conversion rule (from `c` to `i`) before moving the data to `f2`. If

the conversion is successful, the data is moved. If the conversion is unsuccessful, the system throws a runtime error. Because the field `f1` has the value `23`, the conversion will be successful because `23` is a valid integer.

Listing 4.4 assigns an initial value of `T3` for the field `f1`. This is a valid value for a character-type field, but what happens if you try to assign this field to an integer-type field?

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE i.
f2 = f1.
```

Listing 4.4 Type Conversion with Invalid Content

In Listing 4.4, the field `f1` has the value `T3`; because the system can't convert this value to a number, it'll throw a runtime error by raising the exception `CX_SY_CONVERSION_NO_NUMBER` (see Figure 4.5) when the statement `f2 = f1` is executed. The runtime error can be avoided by catching the exception, which we'll discuss in Chapter 9.

```
Error analysis
    An exception occurred that is explained in detail below.
    The exception, which is assigned to class 'CX_SY_CONVERSION_NO_NUMBER', was not
     caught and
    therefore caused a runtime error.
    The reason for the exception is:
    The program attempted to interpret the value "T3" as a number, but
    since the value contravenes the rules for correct number formats,
    this was not possible.
```

Figure 4.5 Runtime Error Raised by Code in Figure 4.4

There are two exceptions: a data object of `TYPE t` can't be assigned to a data object of `TYPE d` and vice versa. In addition, assignments between data objects of nearly every different data type are possible.

Conversion Rules

The *conversion rule* for a `TYPE c` source field and a `TYPE i` target field is that they must contain a number in commercial or mathematical notation. There are a few exceptions, however, as listed here:

- A source field that only has blank characters is interpreted with the number 0.
- Scientific notation is only allowed if it can be interpreted as a mathematical notation.
- Decimal places must be rounded to whole numbers.

You can read all the conversion rules and their exceptions by visiting the SAP Help website at *http://help.sap.com*.

Even though the automatic type conversion makes it easy to move data between different types, don't get carried away. Exploiting all the conversion rules to their full extent may give you invalid data. Only assign data objects to each other if the content of the source field is valid for the target field and produces an expected result.

For example, the code in Listing 4.5 will result in invalid data when a `TYPE c` field containing a character string is assigned to a numeric field.

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE n LENGTH 2.
f2 = f1.
```

Listing 4.5 Example L4eading to Data Inconsistency

In Listing 4.5, `f2` is of `TYPE n` and will have the value `03` (`T` is ignored) instead of raising an exception as in the earlier case when `f2` was declared as `TYPE i`.

Type conversions are also performed automatically with all the ABAP operations that perform value assignments between data objects (e.g., arithmetic operations).

It's always recommended to use the correct `TYPE` for the data you plan to process. This not only saves the time of performing type conversions but also allows your ABAP statements to interpret the data correctly and provide additional functionality.

For example, let's say you want to process a date that's represented in an internal format as YYYYMMDD. The following example explains how the `WRITE` statement interprets this data based on the data type. In Listing 4.6, a data object `date` is defined as a `TYPE c` field with `20151130` as the value to represent the date in internal format. When this field is written to the output using the `WRITE` statement, the value is printed as it is in the output. The `WRITE` statement doesn't interpret this value as a date; instead, it's interpreted as a text value.

```
DATA date TYPE c LENGTH 8 VALUE '20151130'.
WRITE date.
```

Listing 4.6 Date Stored in a TYPE c Field

The output of the code in Listing 4.6 will be `20151130`. If instead you declare the data object `date` as `TYPE d` as shown in Listing 4.7, the output will depend on the date format set by the country-specific system settings or the user's personal settings.

```
DATA date TYPE d VALUE '20151130'.
WRITE date.
```

Listing 4.7 Date Stored in a TYPE d Field

SAP allows you to set your own personal date format. For example, if User 1 has his personal date format set as MMDDYYYY, and User 2 has his personal date format set as DDMMYYYY, hen User 1 executes a program with the code in Listing 4.7, the output would be `11302015`, and when User 2 executes the same program, the output would be `30112015`.

Here, the `WRITE` statement can make use of the system settings/user's personal settings to display the output because it can interpret the value as a date. In Listing 4.6, however, the value in the data object `v_date` was interpreted as a character string, so the code outputted the value as is.

User-Defined Elementary Data Types

In ABAP, you can define your own elementary data types based on the predefined elementary data types. These are called *user-defined elementary data types* and are useful when you want to define a few related data objects that can be maintained centrally.

User-defined elementary data types can be declared locally in the program using the `TYPE` keyword, or you can define them globally in the system in the ABAP Data Dictionary.

For example, if you want to create a family tree or a pedigree chart you'd start by defining various data objects to process the names of different family members. The data object declarations would look something like what's shown in Listing 4.8.

```
DATA : Father_Name TYPE c LENGTH 20,
       Mother_Name TYPE c LENGTH 20,
       Wife_Name TYPE c LENGTH 20,
       Son_Name TYPE c LENGHT 20,
       Daughter_Name TYPE c LENGTH 20.
```

Listing 4.8 Data Objects Referring to Predefined Types

Here, it's assumed that the length of a person's name can be a maximum of 20 characters. Later, if you want to extend the length of a person's name to 30 characters, you'll need to edit the definition of all data objects individually.

In Listing 4.8, the source code needs changes in five different places. This may become time-consuming and tedious in larger programs. To overcome this problem, first you can create a user-defined type and then point all references to a person to this data type, as shown in Listing 4.9.

```
TYPES  person TYPE c LENGTH 20.
DATA : Father_Name TYPE person,
       Mother_Name TYPE person,
       Wife_Name TYPE person,
       Son_Name TYPE person,
       Daughter_Name TYPE person.
```

Listing 4.9 Data Objects Referring to User-Defined Types

In Listing 4.9, you first define a user-defined elementary data type `person` using the `TYPE` keyword, and this data type is then used to define other data objects. Here, the data type `person` is based on the elementary `TYPE c`. This way, if you need to extend the length of all data objects referring to a person in your code, you just need to change the definition of your user-defined type rather than edit each individual data object manually.

The basic syntax of the `TYPE` keyword is similar to that of the `DATA` keyword, but with a different set of additions that can be used. Remember that data types don't occupy any memory space on their own and can't be used to store any work data. You always need to create a data object as an instance of a data type to work with the program data.

Data types and data objects have their own namespaces, which means that a data type and data object with the same name in the same program can exist. However, to avoid confusion, we follow certain naming conventions while defining various data types and data objects. For example, global variables are prefaced with `gv_`, local variables with `lv_`, internal tables with `it_`, and so on. These naming conventions are generally set by a company as part of its programming best practices and coding standards.

Complex Data Types

Complex data types are made up of a combination of other data types and must be created using existing data types. Complex data types allow you to work with interrelated data types under a common name. For more information on complex data types, see Chapter 5.

Reference Types

Reference types describe reference variables (data references and object references) that provide references to other objects. For more information on reference types, see Chapter 8. For more information on data references, see Chapter 16.

Internal and External Format of Data

As you've seen, data can be represented in various formats. Say you have a requirement indicating that the user wants to create a document and capture the due date for each document. This data should be updated in the database so that the user can later run a report to pull out all the documents that are due on a particular date.

While creating the document, users would prefer to input the due date in their own personal format. For example, let's assume there are two users, User 1 and User 2, who will be using this application to create documents. User 1 inputs the due date in MM/DD/YYYY format, and User 2 inputs the date in DD-MM-YYYY format. Assume both users have created four documents in total and that the data has been updated in the database. The data in the database table would look like Table 4.4.

Document_Number	Due_Date	Created_By
1001	11/05/2015	USER1
1002	11/06/2015	USER1
1003	05-11-2015	USER2
1004	06-11-2015	USER2

Table 4.4 Data Represented in External Format

Now, the report you want to develop takes the due date as an input to pull out the documents that are due on that particular date. This report will have a `SELECT` statement something like the following to query the database table:

```
SELECT Document_Number FROM db_table WHERE Due_Date EQ inp_due_date.
```

Here, you're comparing the data in the `Due_Date` column of the table with the user-provided date to fetch all the matching document numbers. As you can see, if the user inputs November 5, 2015, as the due date in MM/DD/YYYY format (11/05/2015), then the `SELECT` statement fetches only document number 1001 and ignores document

1003 because the latter doesn't match character-for-character with the given input. If the input is given in DD-MM-YYYY format (05-11-2015), then document 1003 is fetched, and 1001 is ignored. If you input the date in any other format, none of the records are fetched.

Internal and external data formats can help with this issue. If you maintain the internal format of the date as YYYYMMDD, you can convert the user-provided date to the internal format (20151105) and save it to the database. In this way, the data can be saved in the same format consistently. Similarly, you can convert the date to an external format before displaying it to the user so that the user always sees the date in his preferred format.

Following this concept, the data in Table 4.5 can be represented as shown.

Document_Number	Due_Date	Created_By
1001	20151105	USER1
1002	20151106	USER1
1003	20151105	USER2
1004	20151106	USER2

Table 4.5 Dates Represented in Internal Format

Now, when the user inputs a date in your report, it's always converted to the internal format first before querying the database table to fetch the matching records. This allows the user to input the date in any of the valid external formats per his personal choice without worrying about the format the program expects.

Output Length of Data Types

In the previous section, the external format of date included two separators (/ or -). The internal length of a date is eight characters, but to represent the date in an external format requires two extra characters to accommodate the separators. Therefore, the external length should be a minimum of 10 characters. This is determined by the output length of the data type.

If you examine the data object of `TYPE i` in the debugger, you'll see that different lengths have been assigned for **Length** and **Output Length**, as shown in Figure 4.6.

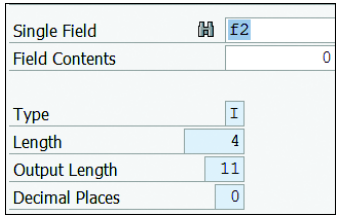


Figure 4.6 Output Length

As shown in Figure 4.6, the data object **f2** is of **Type I** with **Length 4** bytes. However, the **Output Length** is **11** to accommodate the thousands separator for the value.

If you check the same for a data object of **Type D**, you'll see that the **Length** and the **Output Length** are the same—8 bytes—as shown in Figure 4.7.

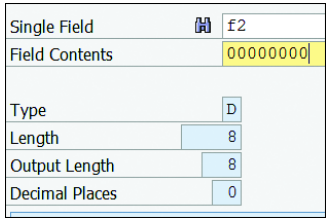


Figure 4.7 Output Length of Type D

If you return to Listing 4.7, you'll see that the system outputted the date without the separators because there was no space in the data object to accommodate the separators. Therefore, it simply converted the date to external format while ignoring the separators. The output length of each data type is predefined and can't be overridden manually.

If you want more control, you can create your own user-defined elementary data types, as described earlier in this section.

4.4.2 Data Elements

As mentioned earlier, user-defined elementary data types can be created for local reusability within the program or for global reusability across multiple programs. The global user-defined elementary types are called *data elements* and are created in the ABAP Data Dictionary.

Data types defined using the TYPE keyword are only visible within the same program that they're created in. If you want to create an elementary user-defined type with global visibility across the system, you can do so in the ABAP Data Dictionary. As you may recall from Chapter 3, the ABAP Data Dictionary is completely integrated into the ABAP Workbench. This allows you to create and manage data definitions (metadata) centrally.

Now, let's create a data element in the ABAP Data Dictionary. Follow these steps to create data element ZCB_PERSON, which is of TYPE c and LENGTH 20:

1. Open the ABAP Data Dictionary via Transaction SE11 in the command bar or by navigating to the SAP menu path, **Tools • ABAP Workbench • Development • ABAP Dictionary**.
2. Select the **Data Type** radio button, provide a name for your data element in the field to the right of the button, and click the **Create** button. Because the data elements are repository objects, they should exist in a customer namespace (i.e., the data element name should start with Z or Y; see Figure 4.8).

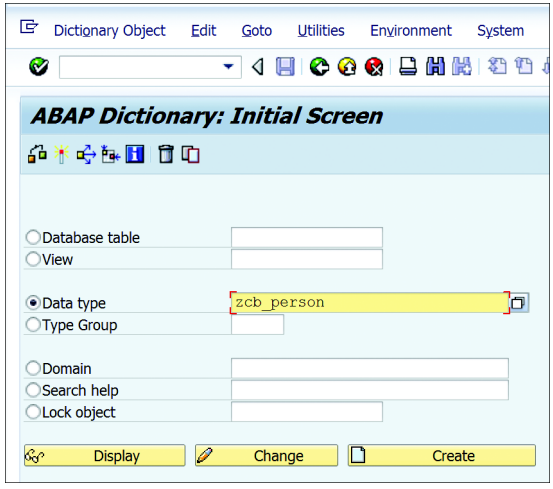


Figure 4.8 ABAP Data Dictionary: Initial Screen

3. The system presents a dialog box (see Figure 4.9) asking you to select the data type you want to create. A data element represents the user-defined elementary data type, so select the **Data Element** radio button, and click **Continue** (green checkmark). (We'll have an opportunity to explore structures and table types in later chapters.)

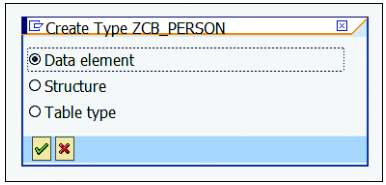


Figure 4.9 Create Type

4. On the next screen (see Figure 4.10), provide a short description for your data type to help others understand its purpose. This short text is also displayed as a title in the F1 help for all screen fields referring to this data element.

   Note that the **Data Type** tab is selected by default, and the **Elementary Type** radio button is preselected. Here, you have two options to maintain the technical attributes for your data element: derive them from a domain or use one of the predefined types. For now, use the predefined elementary data type; we'll explore the domain concept in the next section.

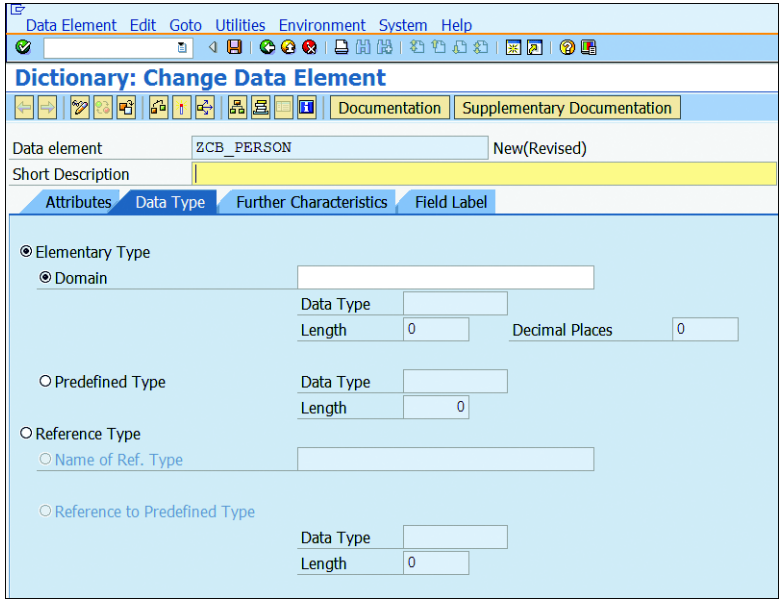


Figure 4.10 Change Data Element Screen

5. Select the **Predefined Type** radio button, and enter the predefined type name in the **Data Type** field. Note that the ABAP Data Dictionary has its own predefined

elementary data types that correspond to the predefined elementary data types in the ABAP programs we discussed earlier.

6. Because we plan to create a character data type, you can enter "CHAR" in the **Data Type** field or select it by using F4 help in the field (see Figure 4.11).

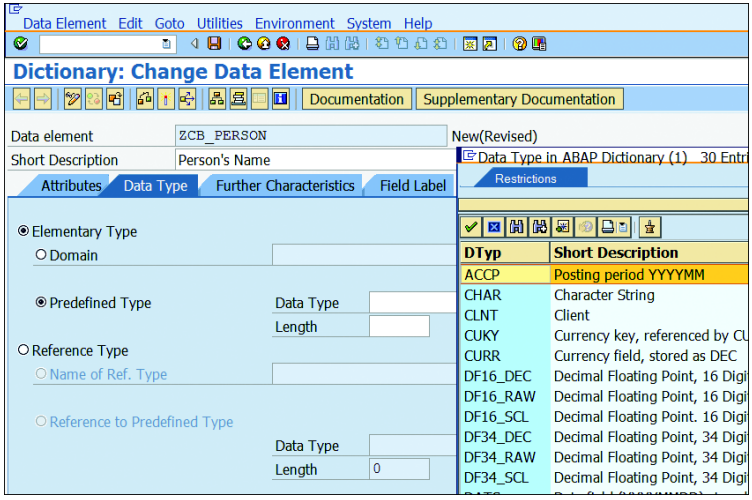


Figure 4.11 Data Type Value List

7. Enter the length of the data type (in this case, "20") in the **Length** field. Click the **Activate** button (see Figure 4.12) or press Ctrl+F3 to activate the data element. Save it to a package, and click **Continue** when an informative message asks you to maintain field labels.

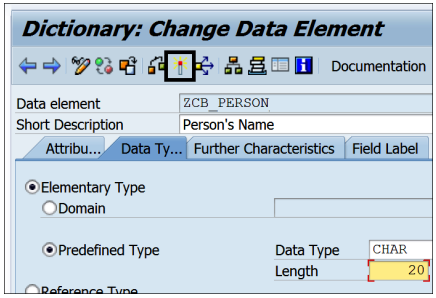


Figure 4.12 Activating a Data Element

8. If the data type supports decimals, you can also maintain the number of decimal places in the **Decimal Places** field.

The data type we just created has global visibility and can be used to define data objects or user-defined data types in any program, as shown:

```
DATA name TYPE ZCB_PERSON.
TYPES user TYPE ZCB_PERSON.
```

If you change the definition of the data element, the changes will be automatically reflected for all the objects referring to that data element. This allows you to maintain the definitions of all related objects centrally. Because data elements are global objects, it's recommended not to change them without analyzing the impact first.

There are many things that can be maintained at the data element level, such as search helps, field labels, and help documentation. The program fields can derive these properties automatically. We'll explore these concepts in detail in Chapter 10.

4.4.3 Domains

A *domain* describes the technical attributes of a field. The primary function of a domain is to define a value range that describes the valid data values for the fields that refer to this domain.

However, if you plan to create multiple data elements that are technically the same, you can attach a domain to derive the technical attributes of a data element. This allows you to manage the technical attributes of multiple data elements centrally, which means you can change the technical attributes once for the domain, and the change will be reflected automatically for all the data elements that use the domain. This is an alternative to changing each individual data element to maintain technical attributes when a predefined elementary data type is selected.

A field can't be referred to a domain directly; it picks up the domain reference through a data element if the domain is attached to the data element. In other words, you always attach a domain to a data element. Figure 4.13 depicts this relationship.

In Figure 4.13, Field 1 refers to Data Element 1, whereas Field 2 and Field 3 refer to Data Element 2. Both Data Element 1 and Data Element 2 are using the same domain to derive technical attributes. This implies that Field 2 and Field 3 are both semantically and technically the same, whereas Field 1 is *semantically* different from Field 2 and Field 3 but is *technically* the same as these two fields.

For example, a sales document number and a billing document number can both be technically the same, such as both being 10-digit character fields. However, the sales document and billing document are semantically different (i.e., their purposes are

different). For such a case, you can use the same domain for both the sales document number and billing document number but a separate data element for each.

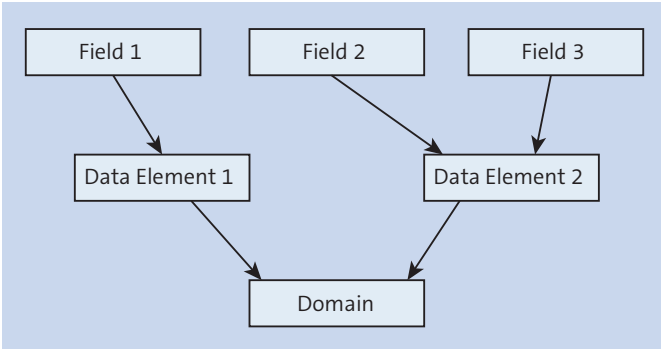


Figure 4.13 Fields, Data Elements, and Domain Relationships

You can also maintain a *conversion routine* at the domain level for automatic conversion to internal and external data formats for fields referring to a domain. We'll discuss this topic more in Chapter 10.

You can take a *top-down* approach or a *bottom-up* approach to create a domain; that is, you either can create the domain separately first and attach it to a data element, or you can create the domain while creating the data element. Let's take a bottom-up approach to create a domain and attach it to the previously created data element `ZCB_PERSON`.

Because domains and data elements have their own namespaces, you can use the same name for both of them. In the following example, you'll create a domain using the same name as the data element created previously in Section 4.4.2; you can use a different name if it becomes too confusing.

The following steps will walk you through the procedure to create a domain in the ABAP Data Dictionary:

1. On the initial ABAP Data Dictionary screen (Figure 4.14), select the **Domain** radio button, input the name of the domain, and click the **Create** button. Domains are also repository objects, so they should exist in a customer namespace.
2. Enter a **Short Description** for the domain. This short description is never seen by end users. It's displayed when searching for domains using F4 help, and it helps other developers understand your domain's purpose.

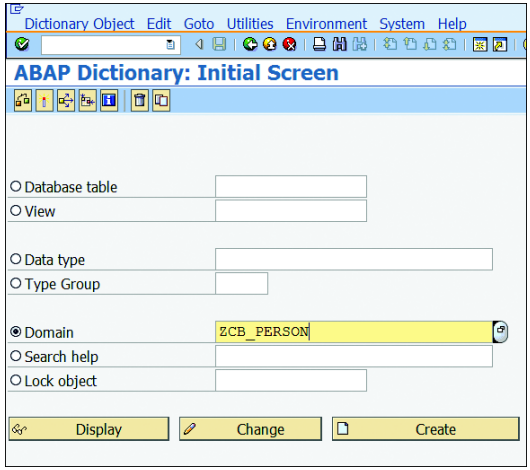


Figure 4.14 Initial Screen of the ABAP Data Dictionary

3. Use the F4 help for the **Data Type** field to select the data type and enter a value for the **No. Characters** field. This value defines the field length (Figure 4.15). Optionally, you can enter the number of **Decimal Places** for numeric types.

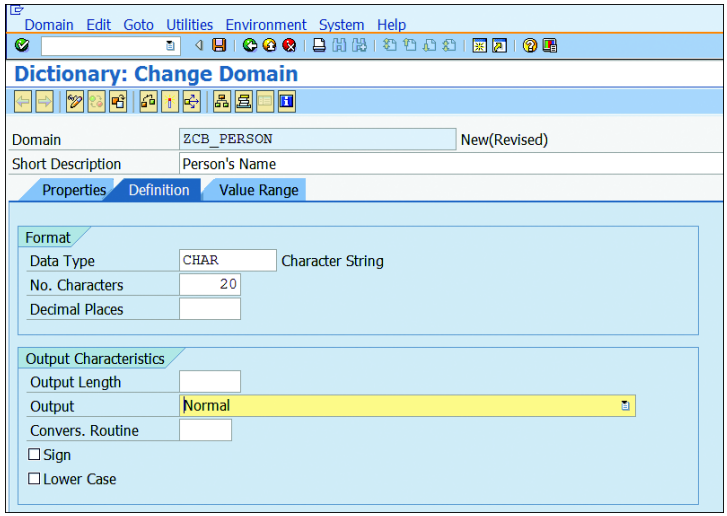


Figure 4.15 Change Domain Screen

4. You can also enter **Output Characteristics** such as **Output Length**, conversion routine (**Convers. Routine**), or disabling automatic uppercase conversion (**Lower Case**)

for screen fields referring to this domain. We'll explore these options further in Chapter 10.

5. Activate the domain. To attach this domain to a data element, open the data element, and select the **Domain** radio button under the **Elementary Type** radio button, as shown in Figure 4.16. Type the name of the domain, and press Enter.

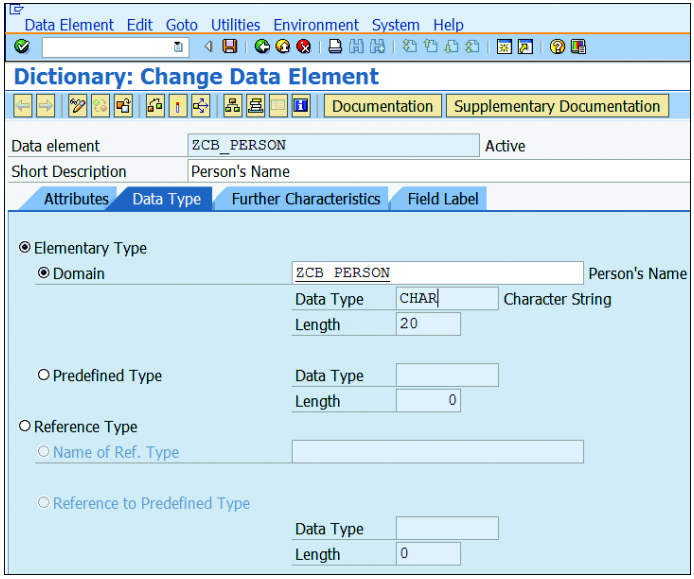


Figure 4.16 Attaching a Domain to a Data Element

6. Notice that the **Data Type** and **Length** are chosen by the system automatically. If the system doesn't set the **Data Type** and **Length** after pressing the Enter key, it means the domain doesn't exist, and an informative message, **No active domain [domain_name] available**, is displayed.

At this point, you can simply double-click the domain name to create it using the top-down approach.

4.4.4 Data Objects

Data objects derive their technical attributes from data types and occupy memory space to store the work data of a program. ABAP statements access this content by addressing the name of the data object. Data objects exist as an instance of data types. Each ABAP data object has a set of technical attributes, such as data type, field length,

and number of decimal places. Data objects are the physical memory units with which your ABAP statements can work. ABAP statements can address and interpret the contents of data objects. All data objects are declared in the ABAP program and are local to the program, which means they exist in the program memory and can be accessed only from within the same program. There is no concept of defining data objects centrally in the system.

Data objects aren't persistent; they exist only while the program is being executed. The life of the data object lasts only as long as the program execution lasts. They are created when the program execution starts and destroyed when the program execution ends.

Before you can process persistent data (e.g., data from a database or a sequential file), you must read the data into data objects first, which can then be accessed by ABAP statements. Similarly, if you want to retain the contents of a data object beyond the end of the program, you must save it in a persistent form. Technically, anything that can store work data in a program can be called a data object.

Let's explore different kinds of data objects that can be defined in your ABAP programs. In the following subsections, we'll look at the different classifications of data objects.

Literals

Literals aren't created using any declarative statements, nor do they have names. They exist in the program source code and are called *unnamed data objects*. Like all data objects, they have fixed technical attributes.

You can't access the memory of a literal to read its content because it's an unnamed data object. This means that literals aren't reusable data objects, and their contents can't be changed. Unlike literals, all other data objects are named data objects, which are explicitly declared in the program.

For example, in the following snippet, `Hello World` and `1234` are literals:

```
WRITE 'Hello World'.
WRITE 1234.
```

There are two types of literals:

- **Numeric literals**
  Numeric literals are a sequence of digits (0–9) that can have a prefixed sign. They don't support decimal separators or notation with a mantissa and exponent.

Examples of numeric literals include the following:

- `+415`
- `-345`
- `400`

The following excerpt shows numeric literals used in ABAP statements:

```
DATA f1 TYPE I VALUE -4563.
WRITE 1234.
F1 = 1234.
```

- **Character literals**
  Character literals are alphanumeric characters in the source code of the program enclosed in single quotation marks (') or back quotes (`).

  For example:

```
'This is a text field literal'.
'1901 CA'.
`This is a string literal`.
`1901 CA`.
```

  Character literals maintained in single quotes have the predefined type `c` and length matching the number of characters. These are called *text field literals*. The ones maintained with back quotes have the predefined type `string` and are called *string literals*. If you use character literals where a numeric value is expected, they are converted into a numeric value. Examples of character literals that can be converted to numeric types include the following:

  - `'12345'.`
  - `'-12345'.`
  - `'0.345'.`
  - `'123E5'.`
  - `'+23E+12'.`

Variables

Variables are data objects for which content can be changed via ABAP statements. Variables are named data objects and are declared in the declaration area of the program. Different keywords, such as `DATA` and `PARAMETERS`, declare different types of variables.

For now, let's explore these two keywords, which define variables, and defer the discussion of other keywords to later chapters.

DATA

The `DATA` keyword defines a variable in every context. You can use this keyword in the global declaration area of the ABAP program to declare a global variable that has visibility throughout the program, or you can use it inside a procedure to declare a local variable with visibility only within the procedure, for example:

```
DATA field1 TYPE i
```

In the preceding statement, a variable with the name `field1` is defined as an integer field using the `DATA` keyword.

> **Inline Declarations**
>
> SAP NetWeaver 7.4 introduced *inline declaration*, which no longer restricts you to define your local variables separately at the beginning of the procedure. You can define them inline as embedded in the given context, helping you to make your code thinner. We'll explore this concept in later chapters.

PARAMETERS

The `PARAMETERS` keyword plays a dual role. It defines a variable within the program context and generates a screen field (selection screen). This keyword is used to create a selection screen for report programs. In these report programs, we present a selection screen for the user to input the selection criteria for report processing, for example:

```
PARAMETERS p_input TYPE c LENGTH 10.
```

In the preceding statement, a 10-character input field with the name `p_input` is defined on the selection screen using the `PARAMETERS` keyword. This field also exists as a variable in the program and is linked to the screen field. The input made on the selection screen for this input field will be stored in the program as the content of the `p_input` variable.

Constants

Constants are named data objects that are declared using a keyword and whose content can't be changed by ABAP statements. The keyword used to declare a constant is

CONSTANT. It's recommended to use constants in lieu of literals wherever possible. Unlike literals, constants can be reused and maintained centrally. The syntax of constants statement mostly matches the data statement. However, with constants statements, it's mandatory to use the addition VALUE to assign an initial value. This value can't be changed at runtime, for example:

```
CONSTANTS c_item TYPE c LENGTH 4 VALUE 'ITEM'.
```

Text Symbols

A *text symbol* is a named data object in an ABAP program that isn't declared in the program itself. Instead, it's defined as a part of the text elements of the program.

A text symbol behaves like a constant and has the data type c with the length defined in the text element. We'll explore text symbols further in Chapter 6.

An example of a text symbol is WRITE text-001. In this statement, a text symbol of the program with the number 001 is written to the output using the WRITE statement. This text symbol is defined separately in the ABAP Editor via the menu path, **Goto • Text Elements • Text Symbols**.

Text symbols are accessed using the syntax text-nnn, where nnn is the text symbol number.

4.5 ABAP Statements

As we discussed earlier, the source code of an ABAP program is made up of various ABAP statements. Unlike other programming languages, such as C/C++ or Java, which contain a limited set of language-specific statements and provide most functionality via libraries, ABAP contains an extensive set of built-in statements. We'll explore many ABAP statements as we progress in this book.

The best way to learn about the various ABAP statements available is to put them in perspective with the requirements at hand. It's beyond the scope of this book to cover all the available ABAP statements, so we'll briefly introduce some popular statements here, and we'll explore these in detail in upcoming chapters:

- **Declarative statements**
  Declarative statements define data types or declare data objects that are used by the other statements in a program. Examples include the following: TYPE, DATA, CONSTANTS, PARAMETERS, SELECT-OPTIONS, and TABLES.

- **Modularization statements**
  Modularization statements define the processing blocks in an ABAP program. *Processing blocks* allow you to organize your code into modules. All ABAP programs are made up of processing blocks, and different processing blocks allow you to modularize your code differently. We'll explore this topic further in Chapter 7.

  Examples include the following: `LOAD-OF-A-PROGRAM`, `INITIALIZATION`, `AT SELECTION SCREEN`, `START-OF-SELECTION`, `END-OF-SELECTION`, `AT USER-COMMAND`, `AT LINE-SELECTION`, `GET`, `AT USER COMMAND`, `AT LINE SELECTION`, `FORM-ENDFORM`, `FUNCTION-ENDFUNCTION`, `MODULE-ENDMODULE`, and `METHOD-ENDMETHOD`.

- **Control statements**
  Control statements control the flow of the program within a processing block. Examples include `IF-ELSEIF-ELSE-ENDIF`, `CASE-WHEN-ENDCASE`, `CHECK`, `EXIT`, and `RETURN`.

- **Call statements**
  Call statements are used to call processing blocks or other programs and transactions.

  Examples include `PERFORM`, `CALL METHOD`, `CALL TRANSACTION`, `CALL SCREEN`, `SUBMIT`, `LEAVE TO TRANSACTION`, and `CALL FUNCTION`.

- **Operational statements**
  Operational statements allow you to modify or retrieve the contents of data objects.

  Examples include `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `SEARCH`, `REPLACE`, `CONCATENATE`, `CONDENSE`, `READ TABLE`, `LOOP AT`, `INSERT`, `DELETE`, `MODIFY`, `SORT`, `DELETE ADJACENT DUPLICATES`, `APPEND`, `CLEAR`, `REFRESH`, and `FREE`.

- **Database access statements (Open SQL)**
  Database access statements allow you to work with the data in the database. Examples include `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MODIFY`.

In this chapter thus far, we've explored the general program structure of an ABAP program, learned basic rules for using ABAP syntax, touched on the use of ABAP keywords and ABAP keyword documentation, and introduced the `TYPE` concept for data types, data elements, and data objects.

In the next section, you'll start creating your first ABAP program using what you've learned so far.

4.6 Creating Your First ABAP Program

Before we explore ABAP basics further, you'll have the chance to create your first program using the ABAP Editor.

You'll need developer access to the SAP system with relevant development authorizations and a developer key assigned to your user ID. Contact your system administrator if the system complains about missing authorizations or prompts you for a developer key when creating the program.

To begin, follow these steps:

1. Open Transaction SE38 or choose **Tools • ABAP Workbench • Development • ABAP Editor**.
2. On the ABAP Editor initial screen, enter the program name, select the **Source Code** radio button, and click the **Create** button. Because this program will be a repository object, it should be in a customer namespace starting with Z or Y, as shown in Figure 4.17.
3. You'll see the **Program Attributes** window (Figure 4.18) to maintain the attributes for your program. *Program attributes* allow you to set the runtime environment of the program.

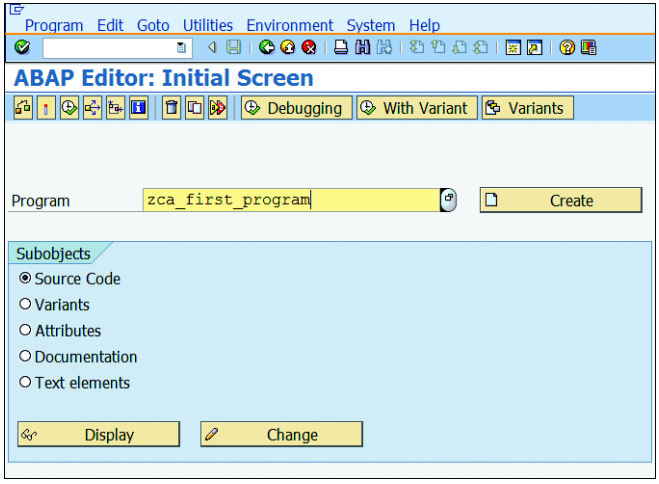


Figure 4.17 ABAP Editor: Initial Screen

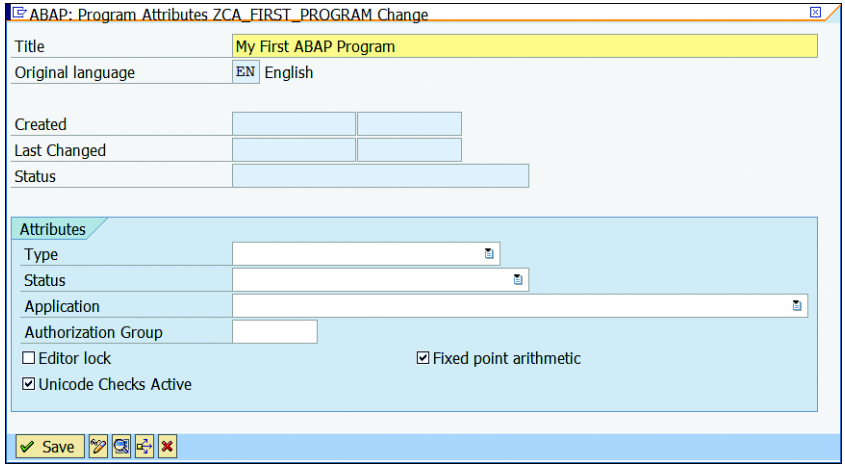


Figure 4.18 Program Attributes Screen

Here, you can maintain a title for your program and other attributes. The **Title** and **Type** are mandatory fields; the others are optional.

Table 4.6 provides an explanation of all the attributes and their significance.

Attribute	Explanation
Type	Allows you to select the type of program you want to create. This is the most important attribute and specifies how the program is executed. It's a mandatory field. From within the ABAP Editor, you can only choose from the following program types: ■ **Executable Program** ■ **Module Pool** ■ **Subroutine Pool** ■ **Include Program** All other program types aren't created directly from within ABAP Editor but are created with the help of special tools such as the Function Builder for function groups or the Class Builder for class pools.
Status	Allows you to set the status of the program development, for example, production program or test program.

Table 4.6 Program Attributes

Attribute	Explanation
Application	Allows you to set the program application area so that the system can allocate the program to the correct business area, for example, SAP ERP Financial Accounting (FI).
Authorization Group	Enter the name of a program group. This allows you to group different programs together for authorization checks.
Logical Database	Visible only when the program type is selected as an executable program. This attribute determines the logical database used by the executable program and is used only when creating reports using a logical database. Logical databases are special ABAP programs created using Transaction SLDB that retrieve data and make it available to application programs.
Selection Screen	Visible only when the program type is selected as an executable program. This attribute allows you to specify the selection screen of the logical database that should be used.
Editor lock	If set, other users can't change, rename, or delete your program. Only you can change the program, its attributes, text elements, and documentation, or release the lock.
Fixed point arithmetic	If set for a program, the system rounds type `p` fields according to the number of decimal places or pads them with zeros. The decimal sign in this case is always the period (`.`), regardless of the user's personal settings. SAP recommends that this attribute is always set.
Unicode Checks Active	Allows you to set whether the syntax checker should check for non-Unicode-compatible code and display a warning message. As of SAP NetWeaver 7.5, the system doesn't support non-Unicode systems, so this option is always selected by default.
Start using variant	Applicable only for executable programs. If you set this attribute, other users can only start your program using a variant. You must then create at least one variant before the report can be started.

Table 4.6 Program Attributes (Cont.)

4. Under the **Attributes** section, provide the following values (see Figure 4.19):
   - **Type**: **Executable program**
   - **Status**: **Test Program**
   - **Application**: **Unknown application**
5. Check the **Unicode Checks Active** and **Fixed point arithmetic** checkboxes.
6. Click the **Save** button when finished.

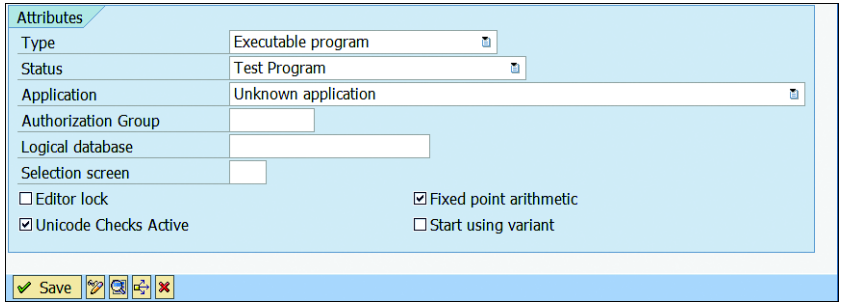


Figure 4.19 Maintaining Program Attributes

7. On the **Create Object Directory Entry** screen (see Figure 4.20), you can assign the program to a package. The package is important for transports between systems. All the ABAP Workbench objects assigned to one package are combined into one transport request.

   When working on a team, you may have to assign your program to an existing package, or you may be free to create a new package. All programs assigned to the package `$TMP` are private objects (local object) and can't be transported into other systems. For this example, create the program as a local object.

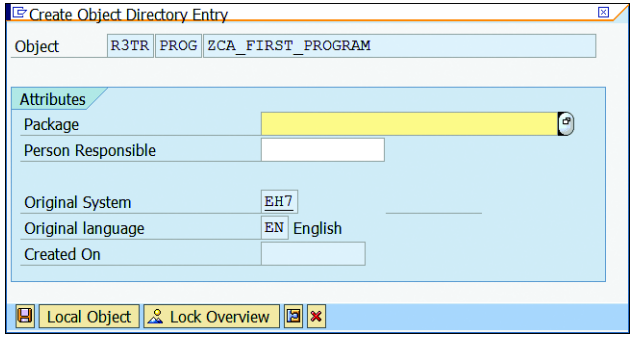


Figure 4.20 Assigning Package

8. Either enter "$TMP" as the **Package** and click the **Save** button or simply click the **Local Object** button (see Figure 4.20).
9. This takes you to the **ABAP Editor: Change Report** screen where you can write your ABAP code. By default, the source code includes the introductory statement to introduce the program type. For example, executable programs are introduced with the `REPORT` statement, whereas module pool programs are introduced with the `PROGRAM` statement.
10. Write the following code in the program and activate the program:

```
PARAMETERS p_input TYPE c LENGTH 20.

WRITE : 'The input was:', p_input.
```

11. In the code, you're using two statements—`PARAMETERS` and `WRITE`. The `PARAMETERS` statement generates a selection screen (see Figure 4.21) with one input field. The input field will be of `TYPE c` with `LENGTH 20`, so you can input up to 20 alphanumeric characters.

    Here, the `PARAMETERS` statement is performing a couple of tasks:

    – Declaring a variable called `p_input`.
    – Generating a screen field (screen element) on the selection screen with the same name as the variable `p_input`. The screen field `p_input` is automatically linked to the variable sharing the same name. Therefore, the data transfer between the screen and the program is handled automatically. In other words, if you enter some data in the `p_input` screen field, it will be automatically transferred to the program and stored in the `p_input` variable. This data in the `p_input` variable can then be accessed by other statements, as you're doing with the `WRITE` statement to print it in the output.

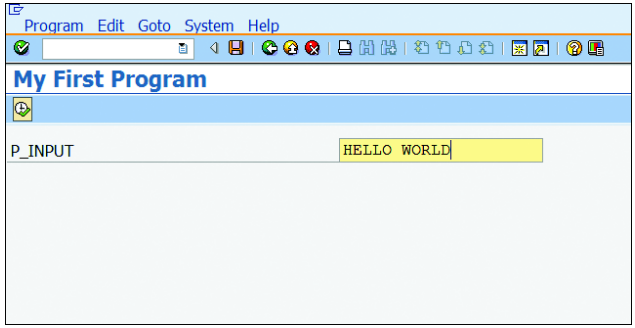


Figure 4.21 Selection Screen

12. Enter a value in the input field, and click the **Execute** button or press F8. This should show you the output or **List** screen, as shown in Figure 4.22.

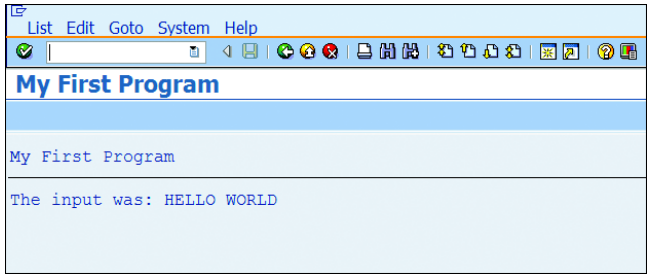


Figure 4.22 List Screen

13. The list screen (output screen) is generated by the `WRITE` statement in the code. With the `WRITE` statement, you're printing the contents of two data objects here. One is a text literal, `'The input was:'`, and the other is the variable `p_input` created by the `PARAMETERS` keyword.

Whatever input was entered on the selection screen was transferred to the program and stored in the variable `p_input`, which was then processed by the `WRITE` statement to display in the output.

You'll have many opportunities to work with different types of screens as we progress through the book.

4.7 Summary

In this chapter, you learned the basics of the ABAP programming language elements. You saw that ABAP programs can only work with data of the same program. Because we typically process various kinds of data in the real world, we explained the different elementary types that are supported by the system to process various kinds of data. Because the basic task of an ABAP program is to process data, this understanding is crucial to process the data consistently.

We also described the syntax to write a couple of statements. We'll explore more statements as we progress through the book. In the final section, you created your first ABAP program.

By now, you should have a good understanding of data types and data objects. However, if you find yourself a little lost, don't worry; the concepts should make more sense as we work through creating more programs in the next chapter. In the next chapter, we'll discuss complex data types and internal tables.

Chapter 5
Structures and Internal Tables

This chapter explores complex data types and lays the foundation for working with data stored in typical database tables.

In the previous chapter, we discussed how to declare and work with individual fields. However, most of the time, we'll be working with data from a group of related fields. For example, when you store a customer's data, you won't store all of it in one field; you'll create multiple fields to store all the different information about the customer: one field to store the customer's name, one for the customer's address, one for the phone number, and so on.

In this chapter, we'll discuss the data types and data objects that let us work with data in the form of an array (a series of objects). Figure 5.1 shows some sample data in a spreadsheet. Here, each field is represented by a column, and each row stores information about a customer.

	A	B	C	D
1	Name	Address	Phone Number	Email ID
2	James	ABC COMPANY Chicago	111-111-111	abc@company.com
3	Ria	XYZ Corp Bangalore	222-222-222	abc@corp.com
4				

Figure 5.1 Data Represented in Rows and Columns

The data in a database table looks similar to what's shown in Figure 5.1. When you want to process this data from a database table in an ABAP program, you need data objects that can store this data in a similar fashion, like a set of fields grouped together to represent a row.

These data types and data objects with individual fields are grouped together into structure types and structures, respectively. We use the term *structure type* to refer to a data type and the term *structure* to refer to a data object. A structure consists of one or more fields. The fields in a structure are called *components*. Each field in a structure

is defined using a *type* or *like* reference, similar to an individual field (variable), which we discussed in Chapter 4, Section 4.4.4.

In Section 5.1, we'll look at different ways to define and work with structures. A structure data object consists of only one row, meaning that it can only store one row of data. When you store a new record in the structure, it replaces the existing record. To process multiple rows of data, we need to use internal tables, as discussed in Section 5.2.

Structures and internal tables are mostly used when working with data from a database table; such data is accessed via Open SQL statements. You'll learn more about relational databases and look at different Open SQL statements in Section 5.3. We'll then look at how to process data from database tables using structures and internal tables in Section 5.4.

It's common, especially for a beginner, for programs to have logical issues that are difficult to detect statically during the syntax check. We'll discuss *debugging* in Section 5.5 to help you become familiar with the debugging tool in order to fix any logical issues with your programs. Debugging also helps you understand the program execution better, and we encourage you to debug all the programs you create during the course of this book.

We'll conclude this chapter with an assignment in Section 5.6, which will reinforce the concepts discussed in this chapter.

5.1 Defining Structures

Structures are either local or global. There are several definitions for both local and global, which we'll cover later. Based on the context, the term *local* can mean a couple of things: If we're talking about a program, local may refer to an object (data object or a data type) that is native to a particular procedure within the program. These objects will be visible within the procedure where they're defined. In this context, a *global* object is the one that has visibility throughout the program; it can be accessed from any procedure or processing block in the program. *Visibility* indicates whether a data object or a data type can be accessed from a particular location in the program code.

For example, as shown in Listing 5.1, you can access the data type `ty_sflight` and the data object `ls_sflight`, but only within the subroutine `data`. Trying to access them

from another procedure (the subroutine `output` in the example) of the same program will result in a syntax error because these objects aren't visible outside their parent procedure. However, if you declare both `ty_sflight` and `ls_sflight` in the global declaration area of the program, they can be accessed from any processing block of the program.

```
FORM data.
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA ls_sflight TYPE ty_sflight.
ENDFORM.

FORM output.
*Both of the lines below will result in a syntax error.
DATA lv_sflight TYPE ty_sflight.
SELECT SINGLE carrid connid fldate FROM sflight
                                   INTO ls_sflight
                                   WHERE carrid EQ 'AA'.
ENDFORM.
```

Listing 5.1 Using a Local Declaration within a Subroutine

If we're talking about an SAP system, then local and global will have different meanings. In such a case, *global* means that the object is visible across the system and can be accessed from any ABAP program. Conversely, *local* means that the object is visible only within a particular program and can't be accessed from other programs.

There are three types of ABAP structures, as depicted in Figure 5.2:

- **Flat structures**
  Components are made of individual fields that refer to elementary types.
- **Nested structures**
  At least one component of a structure refers to another structure.
- **Deep structures**
  At least one table type (a type that defines an internal table) is referenced.

In this section, you'll learn how structures are useful in programs. We'll look at how structure types can be defined both locally in the program and globally in the ABAP Data Dictionary. You'll also learn how data can be stored and accessed with structures. We'll conclude this section by providing some use cases for structures.

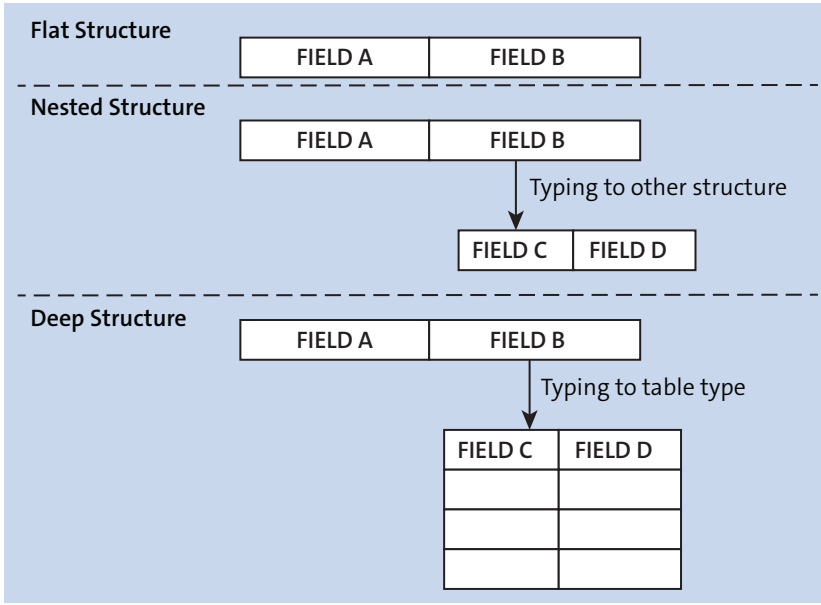


Figure 5.2 Different Structure Types

5.1.1 When to Define Structures

We use structures to process sequential data, such as the data stored in the rows and columns of a database table or a sequential file. Data objects allow you to import data from an external source (e.g., a database table or a sequential file) into your ABAP program, which has to happen before that same data can be accessed by ABAP statements (see Figure 5.3).

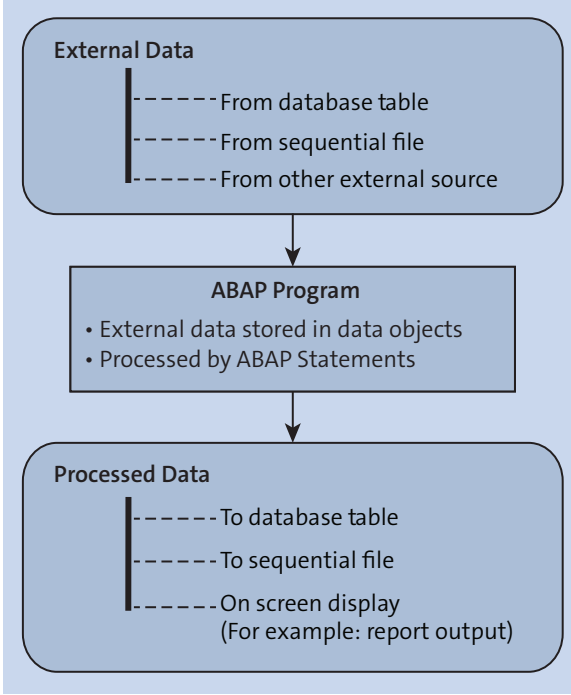


Figure 5.3 Data Processing within ABAP Programs

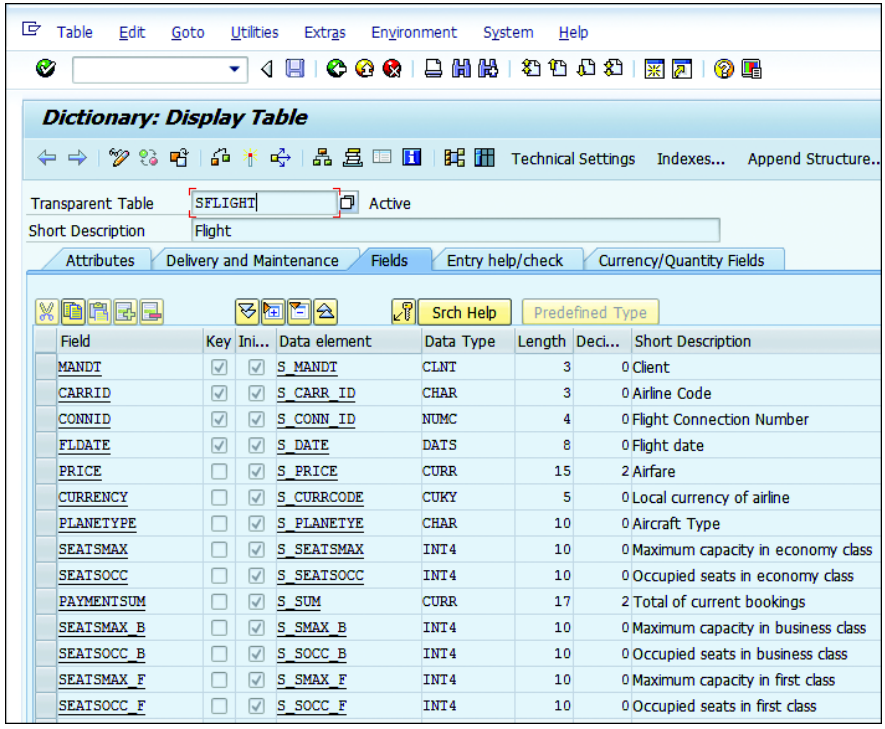
Therefore, whether you need a single variable or a structure with multiple fields depends on the data that you're importing to process in your ABAP program. If you're working with more than one field in a row or a complete row of data, define a structure in your program. If you're working with an individual field or just one field from a row, define a single field in your program.

5.1.2 Local Structures

Let's take the example of table `SFLIGHT` to understand the different ways to define structures in your ABAP program. After you understand how to define a structure, you should be able to put them to use to process data from the database table.

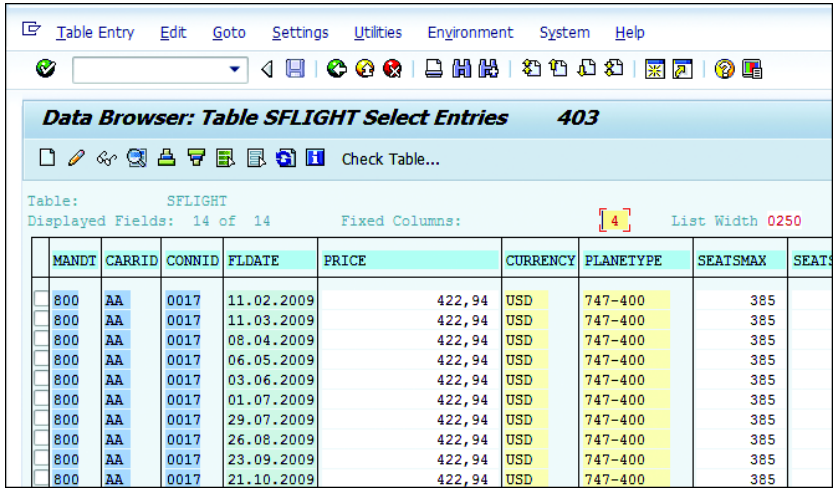
Figure 5.4 shows the table `SFLIGHT` fields.

Figure 5.5 shows the table data in the data browser tool. To open the table, go to Transaction SE11, select the **Database Table** radio button, enter the table name, and click **Display**.



Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
MANDT	☑	☑	S_MANDT	CLNT	3	0	Client
CARRID	☑	☑	S_CARR_ID	CHAR	3	0	Airline Code
CONNID	☑	☑	S_CONN_ID	NUMC	4	0	Flight Connection Number
FLDATE	☑	☑	S_DATE	DATS	8	0	Flight date
PRICE	☐	☑	S_PRICE	CURR	15	2	Airfare
CURRENCY	☐	☑	S_CURRCODE	CUKY	5	0	Local currency of airline
PLANETYPE	☐	☑	S_PLANETYE	CHAR	10	0	Aircraft Type
SEATSMAX	☐	☑	S_SEATSMAX	INT4	10	0	Maximum capacity in economy class
SEATSOCC	☐	☑	S_SEATSOCC	INT4	10	0	Occupied seats in economy class
PAYMENTSUM	☐	☑	S_SUM	CURR	17	2	Total of current bookings
SEATSMAX_B	☐	☑	S_SMAX_B	INT4	10	0	Maximum capacity in business class
SEATSOCC_B	☐	☑	S_SOCC_B	INT4	10	0	Occupied seats in business class
SEATSMAX_F	☐	☑	S_SMAX_F	INT4	10	0	Maximum capacity in first class
SEATSOCC_F	☐	☑	S_SOCC_F	INT4	10	0	Occupied seats in first class

Figure 5.4 Table SFLIGHT Fields



MANDT	CARRID	CONNID	FLDATE	PRICE	CURRENCY	PLANETYPE	SEATSMAX	SEATS
800	AA	0017	11.02.2009	422,94	USD	747-400	385	
800	AA	0017	11.03.2009	422,94	USD	747-400	385	
800	AA	0017	08.04.2009	422,94	USD	747-400	385	
800	AA	0017	06.05.2009	422,94	USD	747-400	385	
800	AA	0017	03.06.2009	422,94	USD	747-400	385	
800	AA	0017	01.07.2009	422,94	USD	747-400	385	
800	AA	0017	29.07.2009	422,94	USD	747-400	385	
800	AA	0017	26.08.2009	422,94	USD	747-400	385	
800	AA	0017	23.09.2009	422,94	USD	747-400	385	
800	AA	0017	21.10.2009	422,94	USD	747-400	385	

Figure 5.5 Table Fields Represented as Columns in Data Browser View

Listing 5.2 defines a structure data type using the TYPES keyword. This structure has three fields from table SFLIGHT. Defining a structure with specific fields from the database table is useful when you want to process certain field's data from the table. All the fields of the structure should have the same technical attributes as the corresponding fields of the database table.

To ensure this, refer the structure components in the program to the same data elements as their corresponding fields in the database table, for example, CARRID TYPE S_CARR_ID.

Here, S_CARR_ID is the data element to which the CARRID field of table SFLIGHT is referring. You may also refer the components of the structure to the table field by using the table reference—for example, CARRID TYPE SFLIGHT-CARRID. Referring the fields to data elements or table fields also ensures that any changes in the definition for the data element or table field will automatically be reflected in the program.

In Listing 5.2, we've defined structure type TY_SFLIGHT using the TYPES keyword with CARRID, CONNID, and FLDATE as components. We've referred the components of the structure type to respective data elements. We've also defined structure ST_FLIGHT using the DATA keyword, referring to the structure type TY_SFLIGHT. The example showing the declaration of structure type TY_SFLIGHT1 uses elementary types to type the components of the structure type.

```
*Notice the chain operator colon(:) after the TYPES keyword,
*without which you will have to use the keyword at the beginning
*of each line and end each line with a period.

TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.

*Since the TYPES keyword created a data type, you can define a
*data object referring to this structure as below.
DATA st_sflight TYPE ty_sflight.

*The same structure type can also be declared while referring the
*fields to elementary types. Note that the type and length of
*each field matches the fields in the table. Matching the type
*and length is very important when using elementary types.
```

```
TYPES : BEGIN OF ty_sflight1,
         CARRID TYPE C LENGTH 3,
         CONNID TYPE N LENGTH 4,
         FLDATE TYPE D,
        END OF ty_sflight1.
```

Listing 5.2 Defining a Structure Type with a Set of Fields

The following shows how to define a structure type to match the ABAP Data Dictionary table:

```
TYPES : ty_sflight TYPE sflight.
```

To define a structure type with all the fields in the table, you can simply refer the structure type to the table. In this example, `ty_sflight` will have all 14 fields of the ABAP Data Dictionary table `SFLIGHT`.

Listing 5.3 shows an example of defining a nested structure.

```
* Defining nested structure type
TYPES : ty_sflight TYPE sflight.
TYPES : BEGIN OF ty_trip,
         name TYPE name1,
         flight TYPE ty_sflight,
        END OF ty_trip.
```

Listing 5.3 Nested Structure Type

In Listing 5.3, we've defined structure type `TY_SFLIGHT` referring to the transparent table `SFLIGHT`. We've also defined a nested structure type `TY_TRIP` that consists of the component `NAME` and the structure type `FLIGHT`.

Listing 5.4 shows an example of defining the structure as a data object using the `DATA` keyword. The syntax is similar to the `TYPES` keyword.

```
*To define a structure as a data object, simply use the
*DATA keyword instead of the TYPES keyword.
DATA : BEGIN OF st_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF st_sflight.
*You can also create a structure by referring to the
```

```
*database table to match the table row type.
DATA st_sflight1 TYPE sflight.
```

Listing 5.4 Defining a Data Object as a Structure

In Listing 5.4, structure ST_SFLIGHT consists of three components, CARRID, CONNID, and FLDATE, whereas structure ST_SFLIGHT1 contains all the fields of the ABAP Data Dictionary table SFLIGHT.

5.1.3 Global Structures

Structures types can be defined globally to have visibility across programs. Global structure types are also used to define the parameter interface for function modules or methods, as well as the screen elements for general screens. You'll learn more about function modules and methods in Chapter 7 and more about dialog programming in Chapter 12.

Global structure types are created in the ABAP Data Dictionary (Transaction SE11). Follow these steps to create a global structure:

1. Enter Transaction SE11.
2. Select the **Data type** radio button on the ABAP Data Dictionary initial screen. Enter the name of the structure. The name should be in a customer namespace, starting with Y or Z. Click the **Create** button (Figure 5.6).

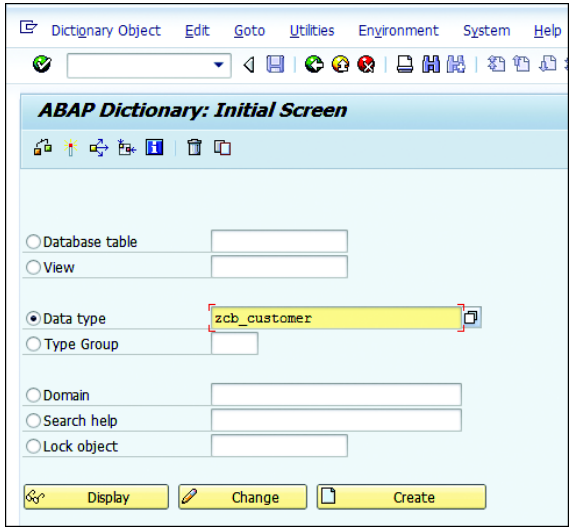


Figure 5.6 ABAP Dictionary Initial Screen

3. In the modal dialog box, select the **Structure** radio button, and click the green checkmark (Figure 5.7).

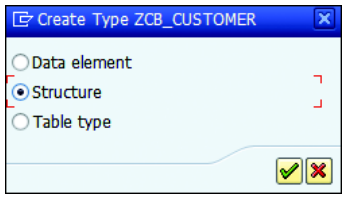


Figure 5.7 Modal Dialog Box to Select Data Type Category

4. On the next screen, enter a **Short Description** for your structure. Maintain the fields as shown in Figure 5.8. Enter the field name under the **Component** column. Select **Types** from the dropdown list under the **Typing Method** column. Enter the data element name under the **Component Type** column, and press Enter.

   For **Component Type**, you can either use an existing data element or create your own data element, as discussed in Chapter 4. If you wish, you can use a predefined elementary type by clicking the **Predefined Type** button.

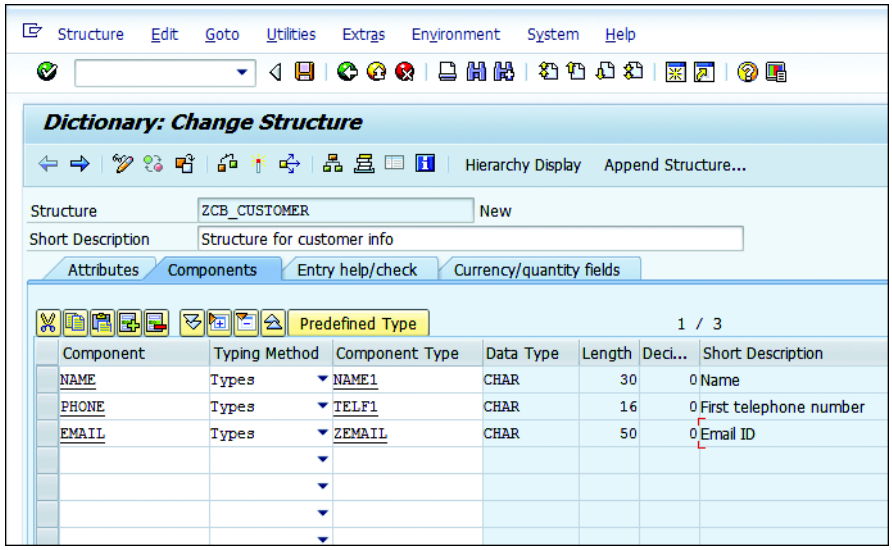


Figure 5.8 Maintaining Structure Type Components

5. Click **Activate** to enter the package name and save it. You can save it as a local object (`$TMP` package) or save it to a transportable package.

6. Upon activation, the system may produce a warning message about a missing enhancement category; however, this warning won't prevent you from activating the structure.

SAP-delivered tables and structures in the ABAP Data Dictionary can be enhanced by adding extra fields through *Customizing Includes* (CI) or *append structures*. We'll learn more about CI and append structures in Chapter 20 when we discuss modifications and enhancements.

The enhancement category setting specifies how a particular table or structure can be enhanced. Using this setting, you can restrict whether the structure can be enhanced by allowing only character type fields, both character and numeric fields, or deep structures.

You can maintain the enhancement category via the **Extras • Enhancement Category** menu path (Figure 5.9). You can refer to the online documentation by using the **Information** button.

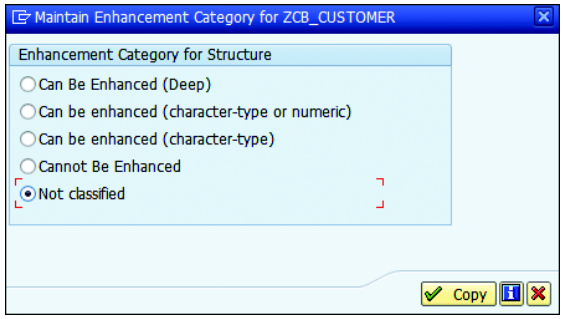


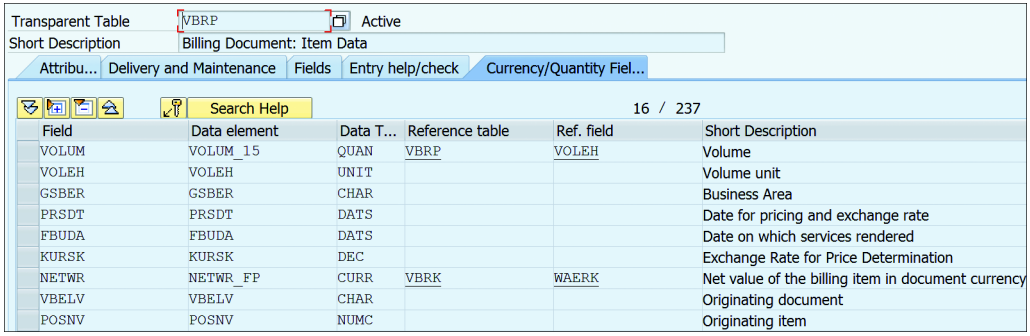
Figure 5.9 Enhancement Category

If your structure has a currency (type `CURR`) or quantity (type `QUAN`) fields, you should maintain reference fields in the **Currency/Quantity Fields** tab (as shown in Figure 5.10). A currency field should refer to a currency key field (type `CUKY`), and a quantity field should refer to a **Unit of Measure** field (type `UNIT`). The reference fields can exist in the same table or another table.

Maintaining the reference fields for currency and quantity fields is mandatory, and the system uses this reference to insert the relevant currency or unit when the field is outputted. If the reference field exists in the same table, then the system can use the reference to apply the correct formatting to the output automatically. While providing the reference fields, enter the reference table name and reference field name in the respective columns.

Figure 5.10 shows the reference fields as maintained in billing document item table VBRP. The VOLUM field of table VBRP is of type QUAN and is referenced to field VOLEH of the same table VBRP, which is of type UNIT. Similarly, the NETWR field of table VBRP is of type CURR and is referenced to the WAERK field of table VBRK, which is of type CUKY.

If the reference fields aren't maintained for currency and quantity fields of your table/structure, you won't be able to activate the table or structure in the ABAP Data Dictionary.



Transparent Table VBRP Active
Short Description Billing Document: Item Data
Attribu... | Delivery and Maintenance | Fields | Entry help/check | Currency/Quantity Fiel...
Search Help 16 / 237

Field	Data element	Data T...	Reference table	Ref. field	Short Description
VOLUM	VOLUM_15	QUAN	VBRP	VOLEH	Volume
VOLEH	VOLEH	UNIT			Volume unit
GSBER	GSBER	CHAR			Business Area
PRSDT	PRSDT	DATS			Date for pricing and exchange rate
FBUDA	FBUDA	DATS			Date on which services rendered
KURSK	KURSK	DEC			Exchange Rate for Price Determination
NETWR	NETWR_FP	CURR	VBRK	WAERK	Net value of the billing item in document currency
VBELV	VBELV	CHAR			Originating document
POSNV	POSNV	NUMC			Originating item

Figure 5.10 Reference Fields Maintained for Currency and Quantity Fields in Table VBRP

5.1.4 Working with Structures

Now that you know how to define both local and global structures, let's look at how to work with them in ABAP programs. There are four possible ways to define a structure data object in a program:

- Refer to a local structure type defined in the program using the TYPES keyword (see Listing 5.5).
- Refer to a global structure type.
- Refer to an ABAP Data Dictionary table.
- Directly declare the data object using the BEGIN OF…END OF syntax for the DATA keyword.

Structures behave similarly to single fields, except for how you access the components of a structure. To access the field of a structure, you need to give the structure name field reference, as shown in the example in Listing 5.5. Note that structures can store only one row at a time. If you store a new record, it will replace the existing record.

The example code in Listing 5.5 defines a flat structure TY_SFLIGHT and a nested structure TY_TRIP. Corresponding data objects ST_SFLIGHT and ST_TRIP are defined that refer to these structures, respectively. The comment lines in the code explain how the components of the structures are accessed using ABAP statements.

```
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.
TYPES : BEGIN OF ty_trip,
         name TYPE name1,
         flight type ty_sflight,
        END OF ty_trip.

DATA : st_sflight TYPE ty_sflight,
       st_trip TYPE ty_trip.

*Adressing to the field of a structure type
DATA v_connid TYPE ty_sflight-connid.

*Accessing field of a structure data object
st_sflight-carrid = 'AA'.
st_sflight-connid = '04'.
WRITE st_sflight-connid.

*Accessing field of a nested structure
st_trip-flight-carrid = 'AB'.
```

Listing 5.5 Accessing Fields of Structure

5.1.5 Use Cases

Structures should be used in the following cases:

- Use structures to process multiple fields of data. For example, when you want to process a record, you can define a structure with the fields of the record as components and process the record as a single entity.
- Group related fields into a structure if you need to perform certain actions on the fields together. It's easy to clear the data in all the fields of a structure at once instead of individually clearing the data for each field.

For example, if you have a structure ST_SFLIGHT with all the components of the ABAP Data Dictionary table SFLIGHT, you can clear a record easily by using the statement CLEAR ST_SFLIGHT rather than using the CLEAR statement for each individual field.

- Even though it's possible to select data from a database table row into individual fields, it's always recommended to use structures when processing data from database tables.
- Structure types also help you keep the parameter interface (signature) of a procedure (e.g., function module or method) simple by grouping relevant fields into one structure as opposed to maintaining all fields individually in the interface.

5.2 Internal Tables

Internal tables are dynamic data objects that can store multiple rows of data with a common row type. In fact, internal tables are the only data objects that can store multiple rows of data at a time. We use internal tables when we need to work with mass data. Internal tables can be compared to two-dimensional arrays.

Working with internal tables is a bit different from other data objects. In this section, we'll look at how to define, store, and access data with internal tables. We'll discuss how to define internal types and the different types of internal tables available. We'll then provide steps for working with internal tables before discussing the use of control break statements.

5.2.1 Defining Internal Tables

Listing 5.6 shows the basic syntax for declaring an internal table. Here, the data object IT_SFLIGHT is declared as an internal table with the line type as TY_SFLIGHT. The *line type* of a table specifies the fields that make up a row. The TABLE OF addition to the type specification defines an internal table.

```
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight.
```

Listing 5.6 Declaring Internal Table

Because an internal table consists of multiple rows, you can't work with the data of an individual row directly. To process the data from an internal table, you need a structure data object, called the work area. The *work area* of an internal table is a structure that can store one record at a time and has the same `Line` type as the internal table. This allows you to read an individual row from an internal table into the work area and process it.

Listing 5.7 shows the work area `wa_sflight` declaration, referring to the same `Line` type as the internal table `IT_SFLIGHT`.

```
TYPES : BEGIN OF ty_sflight,
          CARRID TYPE S_CARR_ID,
          CONNID TYPE S_CONN_ID,
          FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight,
       wa_sflight TYPE ty_sflight.
```

Listing 5.7 Defining Internal Table Work Area

Figure 5.11 depicts the usage of a work area. Because the work area can store only one record, you can read a single row from the internal table or loop through all rows of the table one row at a time.

To read a single record from the internal table, use the `READ` statement, and to process more than one record, use the `LOOP` statement. The `READ` and `LOOP` statements copy the data from the internal table to the work area.

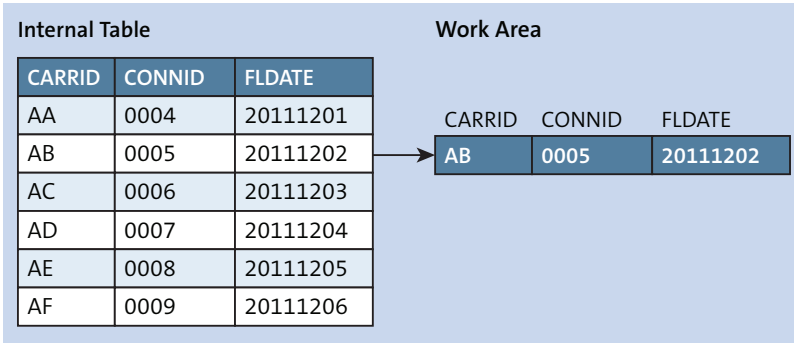


Figure 5.11 Internal Table and Work Area

There are a couple of ways to access a single record from an internal table via the READ statement. You can use an index (index access) or read the table by comparing the value of a field (or a set of fields) of the row (key access). Note that not all internal tables support index access, which we'll discuss in the next section.

Listing 5.8 shows example code to read an internal table using index access.

```
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight,
       wa_sflight TYPE ty_sflight.
SELECT carrid connid fldate FROM sflight INTO TABLE it_sflight.

*Reading a single record from the internal table into work area
*using index access
READ TABLE it_sflight INTO wa_sflight INDEX 2.
```

Listing 5.8 Using Index Access

Listing 5.8 defines a structure type TY_SFLIGHT and an internal table IT_SFLIGHT using the TABLE OF addition to the DATA keyword. The line type of the internal table IT_SFLIGHT is defined as the structure type TY_SFLIGHT. We used a SELECT statement to fetch the data from database table SFLIGHT into the internal table IT_SFLIGHT.

Because the internal table can store multiple records, the system will transfer all the records from the database table SFLIGHT into the internal table IT_SFLIGHT. A SELECT statement should always be restricted using a WHERE clause, but we ignored that rule in this example for simplicity. You'll learn more about Open SQL statements in Section 5.3.

In Listing 5.8, we're reading a single record from the internal table using the READ statement and specifying the index using the INDEX addition. In this example, the second record of the internal table will be copied to the work area.

To read an internal table record using *key access*, use the addition WITH KEY, and list the components of the table to be matched with a value using the = operator as shown in the example ahead. You can list multiple components separated by a space:

```
READ TABLE it_sflight INTO wa_sflight WITH KEY connid = 'AA'.
```

To process all the records of an internal table one row at a time, use the LOOP statement:

```
LOOP AT it_sflight INTO wa_sflight.

ENDLOOP.
```

We can restrict the loop to process specific records using the index:

```
LOOP AT it_sflight INTO wa_sflight FROM 1 to 5.

ENDLOOP.
```

We can also restrict the loop using a key:

```
LOOP AT it_sflight INTO wa_sflight WHERE carrid EQ 'AA'.

ENDLOOP.
```

5.2.2 Types of Internal Tables

An internal table is defined by its row type, table category, and table key. We'll walk through each of these elements in the following subsections.

Row Type

A *row type* defines the row of an internal table (the fields that make up the row of a table). Any type can be used as a row type, such as an elementary type, structured type, table type, or reference type. For example, an internal table can consist of fields (known as elementary types), a structure, a table type, or even a reference object (e.g., an ABAP class).

Table Category

There are three types of internal tables that can be defined in ABAP programs: standard, sorted, and hashed. The *table category* defines the type of internal table based on how individual rows can be accessed.

In the following subsections, we'll take a close look at the different types of table categories that can be defined.

Standard Tables

Standard tables are managed using a row index. For every record that you add to an internal table, the system creates a linear index. New rows can be either appended to the end of the table using the APPEND keyword (e.g., APPEND wa_sflight TO it_sflight) or inserted at specific positions using the INSERT keyword (e.g., INSERT wa_sflight INTO it_sflight INDEX 3). Individual rows are accessed using the table key or the row index; standard tables support both key access and index access.

You aren't required to specify the primary table key explicitly during table declaration. The primary key can be empty for standard tables. Standard tables only support nonunique primary keys, which means that multiple rows in the same table can have the same key value.

When using the TABLE KEY addition with the READ statement, the system performs a linear search to identify the matching row.

The syntax for declaring a standard table is as follows:

```
DATA internaltable_name TYPE TABLE OF [or STANDARD TABLE OF] structure_
reference
```

The TYPE TABLE OF and TYPE STANDARD TABLE OF additions both define standard tables. internaltable_name is the name of the internal table, and structure_reference is a reference to the structure type.

In the statement

```
DATA it_sflight TYPE TABLE OF ty_sflight.
```

the internal table IT_SFLIGHT is defined as a standard table with the line type TY_SFLIGHT. Here, TY_SFLIGHT is a structure type. The TABLE OF addition is a short form to define a standard table. We can also use the long form STANDARD TABLE OF to define the standard table, as shown:

```
DATA it_sflight TYPE STANDARD TABLE OF ty_sflight.
```

Sorted Tables

Sorted tables are similar to standard tables, except the records are always sorted by the key in ascending order. New records can be inserted or appended only if they are sorted. Like standard tables, sorted tables are also managed via a row index, so you can read the individual rows by index or key.

Sorted tables support both unique and nonunique primary keys, and defining the primary key is mandatory when declaring the internal table. If the table has a unique key, the key can't be repeated. A table with a nonunique key can contain multiple records for the key. In other words, for a unique key, you can't have multiple records with the same value for the key fields.

For example, if the key of the table is defined as CARRID and CONNID, the table contains a record in which CARRID has the value 'AA', and CONNID has the value '0017', then you can't have a second record in the table with the same values for CARRID and CONNID. The individual fields that form the key can repeat themselves, but the combination of the key fields should be unique. Therefore, we can have another record with CARRID as 'AA' and a different record with CONNID as '0017', but we can't have another single record that contains CARRID as 'AA' and CONNID as '0017'.

When you define a table with a nonunique key, you can have multiple records with the same values for the key and no restrictions apply. You should choose the key carefully based on the data being processed to avoid data inconsistency.

When using the TABLE KEY addition with the READ statement, the system performs a binary search to identify the matching row.

The syntax for declaring a sorted table is as follows:

```
DATA internaltable_name TYPE SORTED TABLE OF structure_reference WITH UNIQUE
[or NON-UNIQUE] KEY key_fields.
```

To define a sorted table, use the SORTED TABLE OF addition. It's mandatory to define the key of the table during a declaration. The key is defined using the WITH UNIQUE KEY or WITH NON-UNIQUE KEY addition to define a unique or nonunique key, respectively. The key of the sorted table can consist of a single field or multiple fields. If the key contains multiple fields, each field is listed after the WITH UNIQUE (or NON-UNIQUE) KEY addition, separated by a space.

For example, the following statement defines a sorted table IT_SFLIGHT with line type TY_SFLIGHT consisting of unique key CARRID and CONNID (assuming CARRID and CONNID are components of structure type TY_SFLIGHT):

```
DATA it_sflight TYPE SORTED TABLE OF ty_sflight WITH UNIQUE KEY carrid connid.
```

In addition, the following statement defines a sorted table IT_SFLIGHT with nonunique key CARRID and CONNID:

```
DATA it_sflight TYPE SORTED TABLE OF ty_sflight WITH NON-UNIQUE KEY carrid connid.
```

Hashed Tables

Unlike standard and sorted tables, *hashed tables* don't have a row index. They are managed using a *hash algorithm*. Because there is no row index, you can't perform any index operations and can only read the individual rows by the key. Hashed tables support only unique keys, and defining the key is mandatory while declaring the internal table.

When using the `TABLE KEY` addition with the `READ` statement, the system doesn't compare the contents of the key fields manually to identify the matching row. Instead, the system uses hash functions to identify the matching row.

The syntax for declaring hashed table is as follows:

```
DATA internaltable_name TYPE HASHED TABLE OF structure_reference WITH UNIQUE
KEY key fields.
```

Use the `HASHED TABLE OF` addition to define a hashed table and list the table key using the `WITH UNIQUE KEY` addition. The following statement defines an internal table `IT_SFLIGHT` as a hashed table with unique key `CARRID` and `CONNID`:

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight WITH UNIQUE KEY carrid connid.
```

Table 5.1 lists some of the key properties of the table categories.

Standard Table	Sorted Table	Hashed Table
■ Contains row index. ■ Accessed using key and index. ■ Only nonunique primary key. ■ Linear search. ■ Lookup time is directly proportional to the number of entries in the table. ■ Primary key can be empty. ■ Rows aren't sorted implicitly but can be sorted explicitly.	■ Contains row index. ■ Accessed using key and index. ■ Both unique and non-unique primary key. ■ Binary search. ■ Lookup time is directly proportional to the number of entries in the table but faster than linear search. ■ Key is mandatory. ■ Rows are always sorted by key in ascending order by default, and no manual sort operations are allowed.	■ No row index. ■ No `INDEX` operations allowed; only key access. ■ Only unique primary key. ■ Hash algorithm. ■ Lookup time is always constant irrespective of the number of entries. ■ Key is mandatory. ■ Rows aren't sorted implicitly but can be sorted explicitly.

Table 5.1 Key Properties of Internal Tables

Any Tables and Index Tables

In addition to the standard table, sorted table, and hashed table categories, there are two other generic table categories—namely, any table and index tables. The generic table categories are used when typing formal parameters of a procedure and field symbols.

The *any table* category covers all table categories; the *index table* category only covers those table categories in which index operations are possible (standard tables and sorted tables). The objects referred to generic table types don't contain *any fields* statically during design time; instead, any fields defined during runtime are based on the definition of the parent object.

The following example shows a field symbol `<fs_table>` defined as a generic type:

```
FIELD-SYMBOLS <fs_table> TYPE ANY TABLE.
```

Because the field symbol `<fs_table>` is defined as the generic type `ANY TABLE`, you can assign any internal table (be it standard, sorted, or hashed) to the field symbol at runtime. The field symbol will then point to the assigned table.

Field symbols are an important concept in dynamic programming; we'll explore them further in Chapter 16.

5.2.3 Table Keys

Internal tables have a primary table key and can support secondary table keys. A *table key* can consist of a list of structure components with a certain sequence or can use `table_line` as a pseudo-component to specify the entire table row as a key, as shown in the following example:

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight WITH UNIQUE key TABLE_LINE.
```

You can also list the individual components of the table as the table key:

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight WITH UNIQUE key CARRID CONNID.
```

You can also specify the standard key's *default key*. This key contains all components that aren't numeric and aren't themselves table types, as shown:

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight WITH UNIQUE DEFAULT KEY.
```

Support for secondary table keys was introduced with SAP NetWeaver 7.0 EHP 2. Previously, each internal table had just one table key (the primary key). When reading an

internal table, you can use either a table key or a free key; that is, you can either read the table using the complete key defined for the table (table key) or read the table using any of the fields of the table (free key).

When a table key is used to read the table, the system performs a linear search for standard tables, performs a binary search for sorted tables, and uses hash functions for hashed tables. However, when using a free key, the system always performs a linear search irrespective of the type of table, thus degrading the performance.

Because using a table key is very restrictive, many programmers use a free key and compromise on the performance. To overcome this problem, SAP introduced secondary keys. You can define secondary keys with different names and use them while processing the internal table. This provides the flexibility of using different keys while ensuring efficiency.

Along with the primary key, an internal table can have up to 15 secondary table keys with different names. With the introduction of secondary keys, what was previously the *table key* has now become the *primary table key,* and a predefined name for it, `primary_key`, has been introduced. Secondary table keys can be hash keys or sorted keys. *Hash keys* must be unique, whereas *sorted keys* can be unique or nonunique. The system creates a secondary table index for each sorted secondary key of an internal table.

Listing 5.9 shows the syntax to define a standard table with a nonunique sorted secondary key and unique hashed key.

```
DATA it_sflight TYPE STANDARD TABLE OF ty_sflight
       WITH NON-UNIQUE DEFAULT KEY
       WITH NON-UNIQUE sorted KEY  by_airline COMPONENTS carrid
       WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.9 Secondary Keys with a Standard Table

> **Note**
>
> Note that defining the primary and secondary keys is completely optional for standard tables.

In Listing 5.9, we defined the internal table `IT_SFLIGHT` as a standard table with a nonunique default primary key that contains two secondary keys, `BY_AIRLINE` and `BY_FLIGHT`. The secondary key `BY_AIRLINE` is defined as a nonunique sorted key with `CARRID`

as the key component. The secondary key BY_FLIGHT is defined as a unique hash key with CARRID and CONNID as the key components.

The syntax to use a secondary key in a READ statement is shown in the following example; here, key_name signifies the name of the secondary key:

```
WITH KEY[or WITH TABLE KEY] key_name COMPONENTS comp1…
```

To read an internal table using the secondary key, specify the name of the secondary key in the READ statement, as shown in Listing 5.10.

```
*Using the secondary key by_airline to read the internal table.
READ TABLE it_sflight INTO wa_sflight WITH TABLE KEY by_airline COMPONENTS
carrid = 'AA'.
```

Listing 5.10 Reading an Internal Table Using a Secondary Key

In Listing 5.10, assuming the internal table IT_SFLIGHT is defined with a secondary key BY_AIRLINE, we're using the READ statement and specifying the name of the secondary key to read the record of the table using the secondary key.

In Listing 5.11, we've defined an internal table IT_SFLIGHT as a sorted table using the pseudo-component table_line as the unique primary key. Two secondary keys, BY_AIRLINE and BY_FLIGHT, are also defined for the table. The secondary key BY_AIRLINE is defined as a nonunique sorted key, and the secondary key BY_FLIGHT is defined as a unique hashed key.

```
DATA it_sflight TYPE SORTED TABLE OF ty_sflight
       WITH UNIQUE KEY table_line
       WITH NON-UNIQUE SORTED KEY  by_airline COMPONENTS carrid
       WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.11 Secondary Keys for a Sorted Table

Listing 5.12 shows an example to define secondary keys for a hashed table.

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight
       WITH UNIQUE KEY table_line
       WITH NON-UNIQUE SORTED KEY  by_airline COMPONENTS carrid
       WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.12 Secondary Keys for a Hashed Table

In Listing 5.12, we defined an internal table IT_SFLIGHT as a hashed table using the pseudo-component table_line as the unique primary key. Two secondary keys, BY_AIRLINE and BY_FLIGHT, are defined for the table. The secondary key BY_AIRLINE is defined as a nonunique sorted key, and the secondary key BY_FLIGHT is defined as a unique hashed key.

Before secondary keys were available, the only way to have the flexibility to read a table by any key while ensuring performance was to sort a standard table by the key and read it using the BINARY search addition.

Listing 5.13 makes the standard table behave like a sorted table with CONNID as the nonunique primary key.

```
SORT it_sflight BY connid.
READ TABLE it_sflight INTO wa_sflight WITH KEY connid BINARY SEARCH.
```

Listing 5.13 Performing a Binary Search on a Standard Table

Secondary keys provide a lot more flexibility. You can use different keys and improve performance while accessing individual rows of a table. However, the ABAP runtime system needs to administer each additional secondary key, so you should use them carefully. You should define a secondary key for an internal table only when the required key or table index is used in processing statements.

Be mindful of the following considerations when using secondary keys:

- Use secondary keys for large internal tables with infrequently changed content. This ensures that the costs for administrating the secondary keys are only incurred when setting up the internal table initially with the data.
- Always avoid secondary hash keys with too many components. If the hash key has too many components, the system has to deal with a high system load for additional hash management. Use sorted keys for secondary keys with a large number of components.
- If you have secondary keys for an index table, the primary index of the table is updated immediately when you insert or delete rows using a secondary key. Because the standard table uses linear search, deleting a row from a standard table using a secondary key can't be optimized.
- You can use secondary keys for small internal tables in some circumstances if you want to ensure unique table entries in relation to particular components. Primary keys don't allow this, especially in the case of standard tables.

- For pure read access, nonunique secondary keys are as quick as unique keys. Use nonunique secondary keys if unique table entries aren't required. Nonunique secondary keys perform better while updating a table and use a lazy update. With a *lazy update*, secondary keys aren't updated immediately after a record is inserted or modified in the table; they're updated when the secondary key is used explicitly the next time after the records were updated or modified in the table.
- You should not use secondary table keys in the following situations:
  - For small internal tables (less than 50 rows). The performance gain for read accesses is much less compared to the increased storage and administration load.
  - With tables that are updated or modified frequently. Each time the table is modified, the secondary keys need to be updated. The loads incurred by updating the secondary keys outweigh the performance benefit for read accesses. In addition, with delayed updates and lazy updates in particular, the system may perform the update during a read access, affecting the performance of the `READ` statement. This defeats the purpose of using secondary keys because the main purpose of secondary keys is to optimize read access.
  - Avoid using secondary hash keys unless you want to ensure the uniqueness of table entries.

5.2.4 Working with Internal Tables

Now that you know how to define internal tables, let's discuss how to perform various actions with them. As discussed earlier, you need to use a work area to add a record to an internal table or to read a record from an internal table. In the following subsections, we'll discuss the different ways you can work with internal tables.

APPEND and INSERT Statements

To add a record to an internal table, use the `APPEND` or `INSERT` statement. An `APPEND` statement always adds the record at the end of the table and is used with standard tables. An `INSERT` statement behaves differently based on the table type. For example, an `INSERT` statement behaves like an `APPEND` statement for standard tables in that it adds a record at the end of the table if no explicit index is specified.

However, with sorted tables, the `INSERT` statement automatically selects the position based on the sort sequence of the table. You need to use caution when using an `APPEND` statement with sorted tables; the system may produce a runtime error if the

new row doesn't match the sort sequence of the table or if the table is defined with a unique key and the new row creates a duplicate entry. For this reason, we recommend always using the INSERT statement with sorted tables. You can't use APPEND statements with hashed tables because they only support INSERT statements.

Listing 5.14 shows an example of using the APPEND statement to add records to a standard table. To add a record to an internal table, first fill the work area with the required data and then append it to the table using the APPEND statement.

```
*Adding a record to internal table
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
       wa_sflight TYPE ty_sflight.

*You can't refer to the table row directly to add a record.
*Fill the work area with the values and add it to the table.
wa_sflight-carrid = 'AA'.
wa_sflight-connid = '0004'.
wa_sflight-fldate = '20140101'.
APPEND wa_sflight TO it_sflight. "Adds the first row to the table

wa_sflight-carrid = 'AB'.
wa_sflight-connid = '0005'.
wa_sflight-fldate = '20140102'.
APPEND wa_sflight TO it_sflight. "Adds second row to the table
```

Listing 5.14 Using an APPEND Statement to Add Records to an Internal Table

Listing 5.15 shows an example using the INSERT statement. Notice the difference in the syntax compared to the APPEND statement. The listing shows the usage of the INSERT statement with and without an explicit index. When an index is supplied with the INSERT statement, the record is added explicitly in that row.

If the index supplied is greater than the line of the table, no record is inserted, and the system field sy-subrc is set to 4. Refer to the box ahead for an explanation of system fields.

```
*The following code inserts the record as last row
wa_sflight-carrid = 'AC'.
wa_sflight-connid = '0006'.
wa_sflight-fldate = '20140103'.
INSERT wa_sflight INTO TABLE it_sflight.
*The following code inserts the record at a specific position based on
*the index.
wa_sflight-carrid = 'AD'.
wa_sflight-connid = '0007'.
wa_sflight-fldate = '20140104'.
INSERT wa_sflight INTO it_sflight INDEX 1
```

Listing 5.15 Adding Records Using an INSERT Statement

The syntax to use the `INSERT` statement is the same for all table types. However, index operations can't be performed on hash keys.

System Fields

System fields are always available in ABAP programs, and they are filled by the runtime system based on the context. You query the system fields in the program to identify the status. For example, you can query the system field `subrc` to learn whether an `INSERT` operation was successful or whether a `SELECT` statement could find a matching record in the database.

System fields are available as variables in the program. However, they should be treated like constants and should never be manipulated. All system fields are available in the ABAP Data Dictionary structure `SYST` and can be accessed in the program using the predefined structure `SY`. For example, to access the system field `subrc` in an ABAP program, use the structure reference `sy` followed by the field name separated by a hyphen, as in `sy-subrc`.

To refer your program data types and data objects to system fields, use the `SYST` structure reference, for example, `DATA lv_index TYPE syst-index`.

Often, you'll perform an `INITIAL` check on system fields to see if a statement was successful, as shown in Listing 5.16.

```
READ TABLE it_sflight INTO wa_sflight WITH KEY CARRID = 'AA'.
IF sy-subrc IS INTIAL.
*******
ELSE.
```

```
*******
ENDIF.
```

Listing 5.16 System Fields

This example checks if the READ statement could successfully find a matching record and accordingly build the program logic. In this case, sy-subrc will have its initial value if the READ statement was successful; otherwise, it will have a value other than its initial value.

Each keyword updates different system fields with different values based on the context. The keyword documentation helps you understand the system field values set by different keywords. To open the keyword documentation, simply press F1 in the ABAP Editor.

For more information on various system fields, visit SAP Help at *http://s-prs.co/*474906.

A system field check should immediately follow the statement for which the status is queried because other statements also may update the system fields.

COLLECT Statement

One more statement is available to add records to an internal table: the COLLECT statement. It behaves a bit differently from the APPEND and INSERT statements. When you try to add a row with a COLLECT statement, the system checks whether the nonnumeric fields of the row exist in the table.

If the nonnumeric fields match, it won't create a new row but simply adds the numeric fields of the row. If the nonnumeric fields don't match, then it creates a new row.

This statement is useful when you need to consolidate data, such as in a supermarket billing application in which the currency and quantity fields are updated when an existing item in the bill is scanned multiple times rather than creating a new entry each time.

Listing 5.17 shows an example of using a COLLECT statement. In this example, a new entry is added in the table for each new item. However, if the item already exists in the table, then only the quantity and amount fields are updated.

```
TYPES : BEGIN OF ty_bill,
          item TYPE c LENGTH 20, "Item name
          qty  TYPE i,           " Quantity
          amt  TYPE p DECIMALS 2,"Amount
```

```
        END OF ty_bill.
DATA : it_bill TYPE STANDARD TABLE OF ty_bill,
       wa_bill TYPE ty_bill.

wa_bill-item = 'soap'.
wa_bill-qty = 1.
wa_bill-amt = 10.
COLLECT wa_bill INTO it_bill. " Creates a new entry

wa_bill-item = 'brush'.
wa_bill-qty = 1.
wa_bill-amt = 20.
COLLECT wa_bill INTO it_bill. " Creates a new entry

wa_bill-item = 'soap'.
wa_bill-qty = 1.
wa_bill-amt = 10.
COLLECT wa_bill INTO it_bill. "Adds the qty and amt
*fields of the first record instead of creating a new record.
```

Listing 5.17 COLLECT Statement

Reading Internal Table Data

You can't access the records of the internal table directly by addressing the name of the internal table; instead, you use the READ statement to read a single record and the LOOP statement to read more than one record.

To read the internal table, you can use a TABLE KEY or FREE KEY. The TABLE KEY compares all the key fields and automatically performs a linear search for standard tables, performs a binary search for sorted tables, and uses the hash algorithm for hashed tables.

If you use FREE KEY instead of TABLE KEY, you can compare any field of the table, and the system will always perform a linear search irrespective of the type of table. A WITH KEY portion of the READ statement allows you to use a free key, while a WITH TABLE KEY portion expects you to provide the table key.

When using the table key, you should specify the fields in the same order as the key. Listing 5.18 shows the syntax to use the READ statement.

```
*Read statement using free key.
READ TABLE it_sflight INTO wa_sflight WITH KEY connid = '0004'.

*READ statement using table key. Assuming the key is defined as
*carrid and connid
READ TABLE it_sflight INTO wa_sflight WITH TABLE KEY carrid = 'AA' connid =
'0004'.

*READ statement using the index. Note that index operations can
*be performed only on standard and sorted tables.
READ TABLE it_sflight INTO wa_sflight INDEX 3.

*Assuming a secondary key is defined as "by_airline"
READ TABLE it_sflight INTO wa_sflight INDEX 3 USING KEY by_airline.
```

Listing 5.18 Syntax to Use a READ Statement

Listing 5.18 demonstrates using the `READ` statement with a free key, table key, and index. When reading an internal table record using an index, the system uses the index of the primary key by default.

If you want to use the index of the secondary key, you can explicitly specify the secondary key name via the `USING KEY` addition. If you specify a primary key after the `USING KEY` addition, then the behavior is similar to using the `READ` statement without the `USING KEY` addition.

The `LOOP` statement can be used to process multiple rows. The system loops between `LOOP` and `ENDLOOP`, reading one row at a time to the work area. Listing 5.19 shows the syntax to use the `LOOP` statement.

```
*Looping all the records of the internal table
LOOP AT it_sflight INTO wa_sflight.
*Statements to process the row can go here
ENDLOOP.
*Looping specific records of the internal table using key
LOOP AT it_sflight INTO wa_sflight WHERE connid EQ '0004'.

ENDLOOP.
*Looping specific records using secondary key. Assuming a
*secondary key "by_airline" is defined with carrid and connid as
*components.
```

```
LOOP AT it_sflight INTO wa_sflight
                   USING KEY by_airline
                   WHERE carrid EQ 'AA'
                   AND connid EQ '0004'.

ENDLOOP.
*Looping specific records using index.
LOOP AT it_sflight INTO wa_sflight FROM 3 TO 5.

ENDLOOP.

*Looping specific records using secondary key index. Note that
*the secondary key should be a sorted key and not hashed key
LOOP AT it_sflight INTO wa_sflight USING KEY by_airline
                                   FROM 3 TO 5.

ENDLOOP.
```

Listing 5.19 Using the LOOP Statement

Listing 5.19 demonstrates different scenarios to use the `LOOP` statement to process the internal table records. You can loop all the records of the internal table or restrict the loop to certain records. The loop can be restricted either by key or index; you can restrict the loop with a key via the `WHERE` clause.

Modifying Records of an Internal Table

In certain scenarios, you may want to modify the records of the internal table. For example, if you're fetching data about a sales order from a database table to an internal table and displaying it on the screen for the user to edit the sales order, then after the user makes any changes to the records on the screen, you may want to modify the corresponding record in the internal table and update it back to the database table.

The `MODIFY` keyword can be used to modify particular rows of an internal table. You need to be careful while using the `MODIFY` statement, however, because it will result in a runtime error if the key doesn't match. You can only modify the nonkey fields and can't change an active key.

Listing 5.20 shows an example of the `MODIFY` statement using both an index and a key.

```
*To modify using index
MODIFY it_sflight FROM wa_sflight INDEX 3.

*To modify using key
MODIFY TABLE it_sflight FROM wa_sflight.
```

Listing 5.20 Performing Modification Operations

When the index is provided, the corresponding row is modified by copying the contents of the work area to the internal table. When using the key, the system scans for the corresponding record in the table and modifies it by copying the contents of the work area to the internal table.

If you want to restrict the fields of the rows that are modified, you can use the TRANSPORTING addition, as shown ahead. The TRANSPORTING addition transports only the specified fields from the work area to the internal table:

```
MODIFY TABLE it_sflight FROM wa_sflight TRANSPORTING carrid.
```

To delete a row from an internal table, you can use the DELETE keyword:

```
DELETE it_sflight INDEX 3.
DELETE it_sflight WHERE connid EQ 'AA'.
CLEAR it_sflight. "To delete all rows
```

5.2.5 Control Break Statements

While processing data in a loop, special control-level processing can be performed via control break statements. The statement block LOOP can contain control structures that trigger the control break statements when the control structure is changed. For example, a control break statement is triggered when the first record is processed in a loop, when a field value in the control structure changes, or when the last record of a loop is processed. Control break statements are introduced with AT and closed with ENDAT.

Four control break statements are available:

- **AT FIRST**
  Triggered in the first loop pass.
- **AT NEW comp**
  Triggered when the value of any field up to comp in the row changes, compared to the previous record.

- **AT END OF comp**
  Triggered when the value of any field up to comp in the row changes, compared to the next record.
- **AT LAST**
  Triggered in the last loop pass.

Assuming internal table IT_SFLIGHT has three components (CARRID, CONNID, and FLDATE—in that order), Listing 5.21 shows the syntax of using various control break statements and why they trigger.

```
LOOP AT it_sflight INTO wa_sflight.
AT FIRST.
*The code here will be executed only in the first loop pass.
ENDAT.
AT NEW carrid.
*The code here will be executed if the value of the field carrid
*is different from the previous loop pass.
ENDAT.
AT NEW connid.
*The code here will be executed if the value of carrid or connid
*is different from the previous loop pass. It not only checks the
*field mentioned in the control structure but checks all
*the fields to the left-hand side in the row.
ENDAT.
AT END OF connid.
*The code here will be executed if the value of either carrid or
*connid will be different in the next loop pass.
ENDAT.
AT LAST.
*The code here will be executed only in the last loop pass.
ENDAT.
ENDLOOP
```

Listing 5.21 Control Break Statements

Control break statements can be maintained in any order in the loop because they're always triggered on the control break and not by the order in which they are maintained in the code.

The code sample in Listing 5.22 shows an example of some control break statements. The code in this example prints the header and item details of a billing document, highlighting how control break statements can be used to control the report output.

```
*Billing document header structure
TYPES: BEGIN OF TY_VBRK,
       VBELN TYPE VBELN_VF, "Document Number
       FKDAT TYPE FKDAT, "Bill Date
       NETWR TYPE NETWR, "Net Value
       KUNRG TYPE KUNRG, "Payer
END OF TY_VBRK.

*Billing document item structure
TYPES: BEGIN OF TY_VBRP,
      VBELN TYPE VBELN_VF, "Document Number
      POSNR TYPE POSNR_VF, "Item Number
      ARKTX TYPE ARKTX, "Description
      FKIMG TYPE FKIMG, "Qty
      VRKME TYPE VRKME, "Sales Unit
      NETWR TYPE NETWR_FP, "Net Value
      MATNR TYPE MATNR, "Material Num
      MWSBP TYPE MWSBP, "Tax amt
  END OF TY_VBRP.

DATA: it_vbrp TYPE TABLE OF ty_vbrp,
      it_vbrk TYPE TABLE OF ty_vbrk,
      wa_vbrk TYPE ty_vbrk,
      wa_vbrp TYPE ty_vbrp.

*Selection screen declaration for user selection criteria
SELECT-OPTIONS: s_vbeln FOR wa_vbrp-vbeln.

START-OF-SELECTION.
*Select Header Data
SELECT  VBELN FKDAT NETWR KUNRG  FROM VBRK
                                 INTO TABLE it_vbrk
                                 WHERE  VBELN in s_matnr.

  "Select Item Data
```

```
  SELECT VBELN POSNR ARKTX FKIMG VRKME NETWR MATNR MWSBP
            FROM vbrp INTO TABLE it_vbrp WHERE vbeln in s_vbeln.


END-OF-SELECTION
*The SORT statement ensures the control break is consistent if
*the table is not a sorted table.
SORT it_vbrp BY vbeln.
LOOP AT it_vbrp INTO wa_vbrp.
   AT FIRST. "Print Column Headings
   WRITE AT: /05 'ITEM',
              15 'DESCRIPTION',
              60 'BILLED QUANTITY',
              80 'UNITS',
             105 'NET VALUE',
             130 'MATERIAL NUMBER',
             150 'TAX AMOUNT'.

     WRITE SY-ULINE.
    ENDAT.

    AT NEW VBELN. " To print header data once per new document
     READ TABLE it_vbrk into wa_vbrk WITH KEY
                                    VBELN = wa_vbrp-vbeln.
     WRITE AT : /5 'Billing Document' LEFT-JUSTIFIED.
     WRITE AT : 30 wa_vbrk-vbeln  LEFT-JUSTIFIED COLOR 2.
     WRITE AT : /5 'Payer' LEFT-JUSTIFIED.
     WRITE AT : 30 wa_vbrk-kunrg LEFT-JUSTIFIED COLOR 3.
     WRITE AT: /5 'BILLING DATE' LEFT-JUSTIFIED.
     WRITE AT: 30  wa_vbrk-fkdat LEFT-JUSTIFIED COLOR 5.
     WRITE AT: /5 'NET VALUE' LEFT-JUSTIFIED.
     WRITE AT: 30 wa_vbrk-netwr LEFT-JUSTIFIED COLOR 6.
    ENDAT.
*Print Item details
    WRITE : /5 wa_vbrp-posnr LEFT-JUSTIFIED,
            15 wa_vbrp-arktx LEFT-JUSTIFIED,
            60 wa_vbrp-fkimg LEFT-JUSTIFIED,
            80 wa_vbrp-vrkme LEFT-JUSTIFIED
           105 wa_vbrp-netwr LEFT-JUSTIFIED,
```

```
          130 wa_vbrp-matnr LEFT-JUSTIFIED,
          150 wa_vbrp-mwsbp LEFT-JUSTIFIED.

*To print document total net value at the end of last item in the
*document. The code in AT END OF will execute before the next new
*document.
   AT END OF vbeln
*The keyword SUM can be used within a control break to
*automatically add up the column.
      SUM.
*Prints total Sum for document
      WRITE : / 105 WA_VBRP-NETWR LEFT-JUSTIFIED.
   ENDAT.

*To identify the last record in the table
   AT LAST.
     WRITE : / 'END OF REPORT'.
   ENDAT.
  ENDLOOP.
```

Listing 5.22 Control Break Example

Listing 5.22 processes the header and item data of a billing document. The header data is fetched from database table VBRK, and the item data is fetched from database table VBRP. We're providing a selection screen to the user via the SELECT-OPTIONS statement, which allows the user to input multiple document numbers that should be processed by the program. You'll learn more about SELECT-OPTIONS in Chapter 6.

Based on the user's input, we fetch the relevant records from the database table and process the records to print in the output using the WRITE statement. We're using the control break statements within the LOOP statement of the item table IT_VBRP to format the output.

For example, we're using the AT FIRST statement to identify the first loop pass to print the column heading. AT NEW is used to identify a new document to read and print the header record. AT END OF is used to identify the last item of the current document to print the total sum for the document. Finally, we used the AT LAST statement to identify the last record of the internal table to print a report closing statement.

The WRITE statement supports various formatting techniques, such as specifying the position on the screen to print the record, using the WRITE AT addition, using / to print

the record in a new line, and so on. We'll explore the WRITE statement further in Chapter 13.

The following rules are applicable when using control break statements:

- The internal table fields should be sorted by the fields used in the control break.
- The internal table can't be modified within the loop.
- The content of a work area specified in the LOOP statement after the INTO addition can't be modified.

When the AT-ENDAT control structure is entered, the contents of the work area are modified as follows:

- The content of the components to which the control break is applied remain unchanged.
- All components with a character-like flat data type that are after the current control key are set to character * in every position.
- All the other components that are after the current control key are set to their initial values.

Figure 5.12 and Figure 5.13 display the contents of a work area before entering the control break and after entering the control break, respectively.

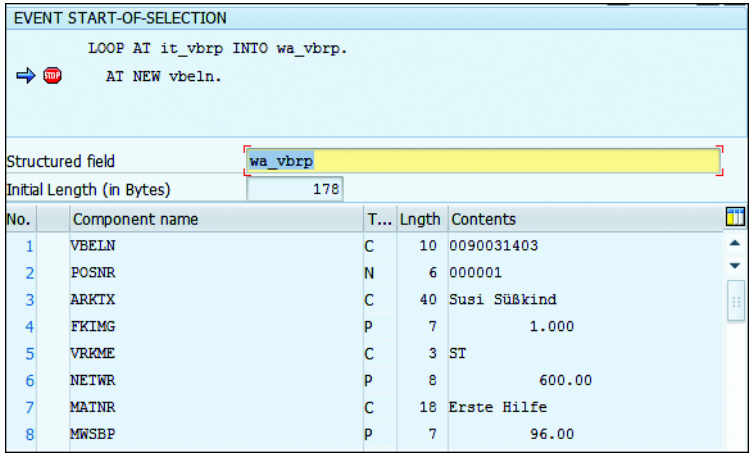


Figure 5.12 Contents of Work Area before Control Break Is Applied as Viewed in Debugger

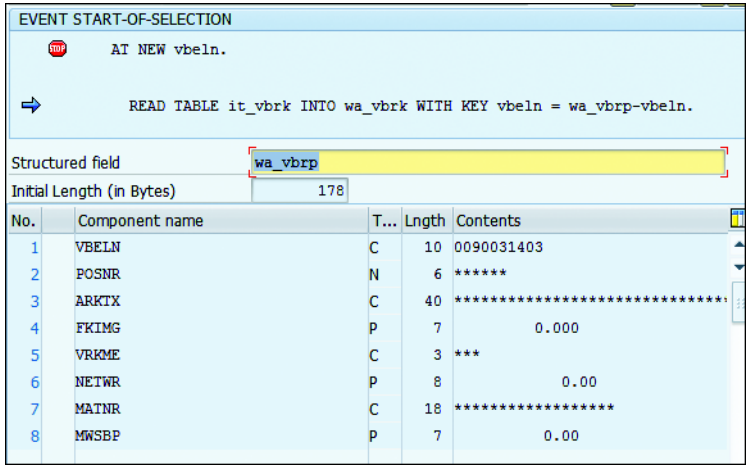


Figure 5.13 Contents of Work Area after Control Break Is Applied as Viewed in Debugger

When the `AT-ENDAT` control structure is exited, the content of the current table row is assigned to the entire work area.

5.3 Introduction to Open SQL Statements

SAP supports different relational database management systems (RDBMSs) such as Oracle, Microsoft SQL, and SAP HANA. Structured Query Language (SQL) is a programming language designed for working with data in an RDBMS.

ABAP supports two types of SQL:

- **Open SQL**
  Open SQL allows you to access database tables and perform certain actions independent of the underlying database the SAP ERP system is using. When executing an Open SQL statement, the database interface of the work process will take care of converting the Open SQL statement to a Native SQL statement, that is, native to the underlying database. This ensures your ABAP code is portable and will work irrespective of the underlying database.
- **Native SQL**
  Native SQL allows you to use database-specific SQL statements in an ABAP program. With a Native SQL statement, you can use database tables that aren't part of the ABAP Data Dictionary. This allows you to integrate data that isn't part of the

SAP ERP system. To use a Native SQL statement in an ABAP program, precede the statement with the `EXEC SQL` keyword and close it with the `ENDEXEC` statement, for example:

```
EXEC SQL.
<native sql statement>.
ENDEXEC.
```

SQL statements are categorized into the following languages:

- **Data Definition Language (DDL)**
  DDL statements are used to define the database structure or schema. Using a DDL statement, you can perform actions such as creating a table, dropping a table, changing the structure of the database table, and so on.

  The following are some examples of DDL statements:

  - `CREATE`: Creates objects in the database.
  - `ALTER`: Alters the structure of the database.
  - `DROP`: Deletes objects from the database.
  - `TRUNCATE`: Removes all records from a table, including all spaces allocated for the records.
  - `RENAME`: Renames an object.
- **Data Manipulation Language (DML)**
  DML statements manage data within schema objects. Using DML statements, you can't alter the database schema, but you can manipulate the data. For example, you can fetch the data from a database table, modify the records of the database table, delete records from the database table, and so on.

  The following are some examples of DML statements:

  - `SELECT`: Retrieves data from a database.
  - `INSERT`: Inserts data into a table.
  - `UPDATE`: Updates existing data within a table.
  - `DELETE`: Deletes all records from a table, leaving the spaces for the records in place.

You can use DML statements with Open SQL, and you can use DDL statements with Native SQL. We generally use the ABAP Data Dictionary to perform tasks that require using DDL statements with Native SQL.

In this section, we'll discuss various key concepts of an RDBMS, such as database design, tables and keys, foreign key relationships, and normalization. We'll also introduce Open SQL statements used to fetch data from database tables.

5.3.1 Database Overview

As discussed in Chapter 2, the database is managed by an RDBMS. A relational database consists of multiple tables that store particular sets of data, and an RDBMS standardizes how the data is stored and processed. In a relational database, data is stored in multiple tables with a relationship between the tables.

The relational database model was first conceived by E. F. Codd in 1969. The model is based on set theory and predicate logic (branches of mathematics). The idea behind the relational model is that the database consists of a series of tables that can be manipulated using a declarative approach (nonprocedural operations).

It's commonly thought that the word *relational* in a relational database pertains to the fact that you *relate* the tables in the database to one another. However, a table is also known as a *relation*, and the word *relational* takes its root from this. In fact, Codd and other database theorists use the terms *relations*, *attributes*, and *tuples* to refer to what developers call *tables*, *columns* (*fields*), and *rows* (or what we'd call *files*, *fields*, and *records* in a physical sense).

In the following subsections, we'll explain how a database is modeled. We'll start by discussing the use of database tables and the significance of table keys. We'll then explore the concept of foreign key relationships to establish relationships between different tables of the database. We'll also discuss the concept of normalization, which helps to simplify database design and avoid redundancy. We'll conclude this section with an introduction to SAP HANA and the many benefits this database provides.

Relational Database Design

A database model to reflect a real-world system takes time and effort to conceive, build, and refine. You need to make decisions about the tables you need to create, what fields they'll contain, and how the tables can be related to each other.

A good database that's designed according to the relational model provides many advantages. A good design makes the data entries, updates, and deletions efficient, and it helps make retrieval and summarization of data easy. Because most data is stored in the database and not in the application, the database should be sufficiently

self-documenting. A good database design should also consider future changes in real-world requirements and should be flexible to allow for changes to the database schema.

Tables and Keys

A *table* is the basic entity in a relational model. Each table should represent an entity in the real world, and these entities can be real-world objects or events. For example, a customer is a real-world object, and the order placed by a customer is an event.

A table consists of rows and columns. The relational model dictates that each row in a table should be unique so that it can be uniquely addressed through programming. The uniqueness of a row is guaranteed by defining a primary key for the table. The primary key of a table consists of one or more fields that uniquely identify a row. Each table can have only one primary key. All fields from which a primary key is drawn are called *candidate keys*. Multiple secondary keys can be drawn from the other fields of the table.

Secondary keys can be unique or nonunique. Secondary keys are generally used to define secondary indexes for the table to speed up data retrieval. Keys can be simple or composite. If a key is made up of one field, it's called a *simple key*; if the key is made up of two or more fields, it's called a *composite key*.

When modeling a table, you should decide which candidate keys form the primary key of the table based on business requirements; there are no hard and fast rules to define the primary key of a table. In his *SQL and Relational Basics* book, Fabian Pascal notes that such a decision should be based on the principles of minimalism (choose the fewest fields necessary), stability (choose a key that seldom changes), and simplicity/familiarity (choose a key that is both simple and familiar to users).

For example, if you have a table to store employee details—such as employee ID, first name, last name, address, telephone number, and zip code—which of these fields will make for a good primary key? Following Pascal's guidelines, the address and telephone number can be ruled out because they change frequently. There's also a chance of the employee's name changing due to marriage or other reasons; plus, names can be misspelled, and there's a high likelihood that two employees will have the same first or last name.

Because the employee ID is unique across the organization, it makes the best candidate for the primary key. Even though there are no fixed rules for choosing a primary key, most developers prefer choosing a numeric primary key because searches on numeric fields are more efficient than those on text fields.

Tip

In many situations, it's recommended to use static numbers, such as employee ID, customer ID, document number, and so on, as primary keys and to avoid descriptive texts.

Foreign Keys and Domains

Even though primary keys are defined in individual tables, there may be a need to define a relationship with different tables of the database using foreign key relations. A *foreign key* is a field in a table that is used to reference a primary key in another table.

Let's look at an example. Table 5.2 shows a customer table in which customer details are stored. The primary key of this table is the *Cust ID* column, which uniquely identifies each row.

Cust ID	Name	Address	Zip Code	Phone
1	Joey	171 CE	110099	99889900
2	James	345 DE	118899	88997788
3	John	563 WE	442299	88993354

Table 5.2 Customer Table

Table 5.3 shows an orders table that stores information about customer orders. The primary key of this table is the *Order ID* column, which uniquely identifies each order.

Order ID	Cust ID	Item	Qty	UoM
2230	1	Pen	10	Pieces
2231	2	Pencil	10	Pieces

Table 5.3 Orders Table

In this example, the Cust ID field of Table 5.3 is considered the foreign key of the table because it can be used to refer to the customer table (see Table 5.2).

It's important that both the primary keys and foreign keys that are used to form a relationship share the same meaning and *domain*. A *domain* defines a possible set of values for a field.

For example, if a valid value for a customer ID can be a number between 1 and 10,000, then the Cust ID fields in both tables (Table 5.2 and Table 5.3) should adhere to this range. By implementing a foreign key check, we can also ensure that the Cust ID field in the orders table (Table 5.3 consists of only the values available in the Cust ID field of the customers table [Table 5.2]).

Relationship

Foreign keys model relationships between real-world entities. Such real-world entities can have complex relationships; for example, one entity may have multiple relations with other entities. In a relational database, relationships are defined between a pair of tables.

These tables can be related in one of three ways:

- **One-to-one relationship**
  Two tables are said to be in a one-to-one relationship if for every row in the first table, there is at most one row in the second table. One-to-one relationships are seldom used to model real-world entities; they are mostly used to split the data into multiple tables due to software limitations.

  For example, you can split data into two tables if the number of fields exceeds the limitation of 249 per table, or you can model two tables with customer data if one of the tables can store more sensitive information about a customer. You have better access to control restrictions for this table compared to the table that stores general information about the customer. The tables in a one-to-one relationship should have similar primary keys.
- **One-to-many relationship**
  Two tables are said to be in a one-to-many relationship if for every row in the first table, there can be zero, one, or multiple rows in the second table, but for every row in the second table there is exactly one row in the first table.

  One-to-many relations are the most commonly modeled. The tables in a one-to-many relation are also called parent-child tables or header-item tables. Common tables with one-to-many relationships in the SAP system include order tables such as `VBAK-VBAP`, which stores header and item details of sales orders, and `EKKO-EKPO`, which stores the header and item details of purchase orders. The one-to-many

relationship is also used to link a base table with a lookup table. For example, in the orders table, you can store a two-character abbreviation for a unit of measure, and this information can be linked to a lookup table in which the descriptions for the abbreviations are maintained.

- **Many-to-many relationship**
  Two tables are said to be in a many-to-many relationship when for every row in the first table, there can be many rows in the second table, and for every row in the second table, there can be many rows in the first table. Many-to-many relations can't be directly modeled in a relational database, and they're typically broken down into multiple one-to-many relations.

Normalization

Normalization is the process of simplifying the design of a database. Normalization answers core questions regarding the number of tables required, what each table represents, the number of fields in each table, and the relationship between tables. Normalization also helps avoid redundancy and anomalies in database design.

Anomalies can occur while inserting, updating, or deleting records from a database table (if the database is poorly designed). For example, if you have a database table as shown in Table 5.4, when a record is updated to change the customer address, all the rows where the customer record exists should be updated. If any rows are missed, then it will lead to data inconsistency. This is called an *update anomaly*.

Item_ID	Customer_ID	Customer_name	Customer_address
1	921	Ryan	12E West End
2	921	Ryan	12E West End
3	728	John	44 Avenue Road

Table 5.4 Update Anomaly Example

If you try to insert a record for a customer that isn't available in the customer master table, it causes an *insertion anomaly*. If you delete item 3 in Table 5.4, it will also delete the record of the customer John because his details are linked to this item only. This is called a *delete anomaly*.

Normalization helps to overcome these anomalies and model a database that is consistent and predictable. Normalizations use the concept of normal forms to assist in

designing an optimal structure. There are three available normal forms: first, second, and third.

The following subsections look at each of these normal forms.

First Normal Form

The first normal form (1NF) dictates that the field of a table should contain atomic values. The field should contain only one value. For example, the data in Table 5.5 doesn't conform to 1NF because the phone number field contains multiple values for the `Emp_id` 2290.

Emp_id	Emp_name	Phone_num
2289	John	99889988
2290	Mark	88998899 99887755

Table 5.5 Employee Not Conforming to 1NF

To make Table 5.5 conform to 1NF, it should be adjusted as shown in Table 5.6 by creating a new record for the second phone number in `Emp_id` 2290.

Emp_id	Emp_name	Phone_num
2289	John	99889988
2290	Mark	88998899
2290	Mark	99887755

Table 5.6 Employee Conforming to 1NF

Second Normal Form

For a table to conform to the second normal form (2NF), it should conform to 1NF, and the nonkey fields of the table should be fully dependent on all the primary key fields and not on the subset of the key fields. For example, in Table 5.7, let's assume that `Employee_ID` and `Department_ID` are the primary key fields. According to 2NF, the nonprimary fields `Employee_name` and `Department_name` should be fully dependent on the complete primary key fields `Employee_ID` and `Department_ID`, not on the subset of the primary key fields.

Employee_ID	Department_ID	Employee_name	Department_name
1122	01	John	Sales
1123	02	Mark	Finance

Table 5.7 Not Conforming to 2NF

That's not the case here; the `Employee_name` field can be identified by the `Employee_ID` field, and the `Department_name` field can be identified fully by the `Department_ID` field. Therefore, the nonprimary fields of this table are only partially dependent on the primary key fields of the table, which isn't allowed in 2NF.

To make Table 5.7 conform to 2NF, it should be split into two tables, as shown in Table 5.8 and Table 5.9. This ensures that there is no partial dependency on primary key fields.

Employee_ID	Department_ID	Employee_name
1122	01	John
1123	02	Mark

Table 5.8 Employee Table

Department_ID	Department_name
01	Sales
02	Finance

Table 5.9 Department Table

Third Normal Form

A table conforms to the third normal form (3NF) if it conforms to 2NF and all nonkey fields of the table are mutually independent. For example, Table 5.10 doesn't conform to 3NF because the field `Total` is dependent on the information in the nonkey fields `Quantity` and `Price`. To make this table conform to 3NF, it's best to remove the `Total` field from the table and handle the calculations in a query or report. This ensures that updating one of the fields of the table doesn't result in any anomalies.

Item_id	Quantity	Price	Total
1	10	10	100
2	20	15	300

Table 5.10 Conforming to 3NF

Every higher normal form is a superset of all lower normal forms. For example, if you design for 3NF, then, it conforms to 1NF and 2NF by default. Normalization helps achieve optimal database design. A good database design takes time, effort, and a good understanding of business requirements. A good design not only makes your applications efficient but also prevents headaches down the line.

SAP HANA

There has been a growing need for real-time analytics over the years, which requires processing huge sets of data. Traditional database systems form data processing speed bottlenecks for large data volumes. SAP addressed this problem with its in-memory database system called *SAP HANA*.

SAP HANA has transformed the relational database industry with its innovations in hardware and software technology to process massive data sets in real time using in-memory computing. SAP HANA combines database, application processing, and integration services on a single platform.

SAP HANA stores compressed data in memory in a columnar format that enables faster scanning and quicker processing. Compared to traditional databases, programs take minutes to process the data from SAP HANA, as opposed to hours when using a traditional database.

The following are some important features to consider with SAP HANA when it comes to ABAP:

- **Parallelism**

  One of the main features of the SAP HANA database is parallelism. SAP HANA takes advantage of the multiple cores of the processor to process information in parallel. Traditional database systems were built decades ago, when only single-core processors existed, and the DRAM was extremely expensive.

  The hardware and computing power of systems has changed tremendously over the years. SAP designed SAP HANA from the ground up to take advantage of modern

hardware capabilities. Typically, an SAP HANA system performs around 3.5 billion scans per second per core and around 12 to 15 million aggregations per second per core. This enables you to perform real-time analytics and process huge sets of data on the fly.

- **Code pushdown**
  The SAP HANA database allows you to push certain data-intensive operations to the database using a technique known as *code pushdown*. For example, if you want to calculate the sum of all invoices, you can use an aggregate function on the database instead of fetching all the invoice data into the application server and calculating the sum in an internal table loop. This can be achieved using ABAP core data services (CDS) views.
- **ABAP CDS views**
  ABAP CDS views aren't persistent objects, but instead projections of a database entity. An ABAP CDS view is created as a design-time file in the repository and exists as an ABAP Data Dictionary object once activated. This allows you to access the CDS view using Open SQL and to use aggregate functions.

5.3.2 Selecting Data from Database Tables

You can use internal tables to process data from a database table, allowing you to store multiple rows of data at once. In this section, we'll explore a few Open SQL statements that can be used to process data from a database.

The `SELECT` keyword fetches data from a database table. You can select a single record or multiple records, and you can select the data from the database into a structure or an internal table of your ABAP program. You can either select the complete row (all fields) of a database table or select only specific fields.

> **UNION Addition**
>
> As of SAP NetWeaver 7.5, the `UNION` addition can be used to create a union between the result sets of two `SELECT` statements.

When selecting data from a database, the row of the data object should be identical to the row of the database table. To ensure this, refer your data object to the table when selecting the complete row. If you need to select only certain fields from the row, make sure to define the fields in the same order in which you plan to select them. The

SELECT statement supports an INTO CORRESPONDING FIELDS addition if the order of fields in your data object doesn't match the order of the selection.

Listing 5.23 shows the usage of a SELECT SINGLE statement to select a single row from the database table. If more than one row satisfies the WHERE clause, the first matching row is fetched.

```
TYPES: BEGIN OF ty_vbrk,
         VBELN TYPE VBELN_VF, "Document Number
         FKDAT TYPE FKDAT, "Bill Date
         NETWR TYPE NETWR, "Net Value
         KUNRG TYPE KUNRG, "Payer
       END OF ty_vbrk.

DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
       wa_vbrk TYPE ty_vbrk.

*Selecting a single row of specific fields from table vbrk.
SELECT SINGLE vbeln fkdat netwr kunrg FROM vbrk
                                      INTO wa_vbrk
                                      WHERE vbeln EQ '9000'.
```

Listing 5.23 Using SELECT SINGLE

Listing 5.24 uses SELECT…ENDSELECT to fetch all matching rows, one row at a time. The system loops between SELECT…ENDSELECT, transferring one row at a time to structure wa_vbrk. You can process the contents of wa_vbrk in each loop by writing the statements before ENDSELECT. This is an efficient way to process the records from the database table if you plan to use a database table only once during program execution.

However, if you plan to reuse the data multiple times during the life of the program execution, we recommend using internal tables, which allow you to fetch the data once into your program and access it subsequently multiple times, as opposed to accessing the database every time you need the same data.

```
*Selecting all matching rows of specific fields from the table
*vbrk using a structure.
*Here the system fetches one row at a time and loops between
*select…endselect. The row can be processed between SELECT…ENDSELECT.
SELECT vbeln fkdat netwr kunrg FROM vbrk
                               INTO wa_vbrk
```

```
                                WHERE vbeln EQ '9000'.

ENDSELECT.
```

Listing 5.24 Using SELECT...ENDSELECT

Listing 5.25 shows using an internal table in a SELECT statement, also known as performing an *array fetch*. After the SELECT statement is executed, you can access the data from the internal table as discussed previously.

```
*Selecting all matching rows of specific fields from the table
*vbrk using an internal table.
SELECT vbeln fkdat netwr kunrg FROM vbrk
                               INTO TABLE it_vbrk
                               WHERE vbeln EQ '9000'.
```

Listing 5.25 Selecting Data to Internal Table

Listing 5.26 shows selecting the fields of the complete table row. Notice the difference in data object declaration and the usage of * as the SELECT statement.

```
DATA : it_vbrk TYPE STANDARD TABLE OF vbrk,
       wa_vbrk TYPE vbrk.

*Selecting a single row of all fields from table vbrk.
SELECT SINGLE * FROM vbrk
                INTO wa_vbrk
                WHERE vbeln EQ '9000'.
*Selecting all matching rows of all fields from table vbrk
*using a structure.
SELECT * FROM vbrk
         INTO wa_vbrk
         WHERE vbeln EQ '9000'.
ENDSELECT.
*Selecting all matching rows of all fields from table vbrk
*using an internal table.
SELECT * FROM vbrk
         INTO TABLE it_vbrk
         WHERE vbeln EQ '9000'.
```

Listing 5.26 Selecting All Fields of a Row

5.3.3 Selecting Data from Multiple Tables

The syntax we've discussed so far allows you to select data from a single table. However, in real-world applications, data is spread across multiple tables with foreign keys.

To select data from tables that are in a foreign key relationship, you can use *joins* or use the FOR ALL ENTRIES addition with the SELECT statement. The FOR ALL ENTRIES addition allows you to select the data from the database table and compare it to the data in an internal table that was previously selected from another database table. This is efficient if you're processing data from two tables.

Listing 5.27 shows the syntax to use for all entries. Here, the data is first selected from database table VBRK into internal table IT_VBRK; then the data in table IT_VBRK is compared to select the data from database table VBRP.

When using FOR ALL ENTRIES, make sure the internal table used for comparison isn't empty. Otherwise, the system will execute the SELECT statement without restriction. Notice the IF condition used in the listing.

```
SELECT * FROM vbrk
                  INTO TABLE it_vbrk
                  WHERE vbeln EQ p_vbeln.
IF sy-subrc IS INITIAL.
SELECT * FROM vbrp
                  INTO TABLE it_vbrp
                  FOR ALL ENTRIES IN it_vbrk
                  WHERE vbeln EQ it_vbrk-vbeln.
ENDIF.
```

Listing 5.27 Using FOR ALL ENTRIES

Alternately, two types of *joins* allow you to select data from multiple database tables:

- **Inner joins**
  Inner joins require an entry with the KEY of one table in the second table to be extracted to an internal table.
- **Outer joins**
  Irrespective of whether an entry exists or not in the second table, outer joins still extract data from the first table.

Listing 5.28 highlights the difference between an inner and outer join. If the internal table has three fields—FLD1, FLD2, and FLD3—the result for an inner join and outer join are shown in Listing 5.28.

```
Table A                    Table B
Field 1      Field 2       Field 1       Field 3
X               100            X           ABC
Y               200            X           DEF
Z               300            Z           WXY


Inner Join:
-----------
FLD1     FLD2     FLD3
--------------------
X     100     ABC
X     100     DEF
Z     300     WXY


Outer Join:
-----------
FLD1     FLD2     FLD3
--------------------
X     100     ABC
X     100     DEF
Y     200         "No entry for Key Y in table B so FLD3 is blank
Z     300     GHI
```

Listing 5.28 Inner Join versus Outer Join

Listing 5.29 shows an example using an inner join. Here, we join the columns `carrname`, `connid`, and `fldate` of the database tables `scarr`, `spfli`, and `sflight` using two inner joins. This creates a list of flights from `p_cityfr` to `p_cityto`. An alias name is assigned to each table.

```
PARAMETERS: p_cityfr TYPE spfli-cityfrom,
            p_cityto TYPE spfli-cityto.

TYPES: BEGIN OF wa,
         fldate TYPE sflight-fldate,
         carrname TYPE scarr-carrname,
         connid   TYPE spfli-connid,
```

```
       END OF wa.

DATA itab TYPE SORTED TABLE OF wa
          WITH UNIQUE KEY fldate carrname connid.

SELECT c~carrname p~connid f~fldate
       FROM ( ( scarr AS c
         INNER JOIN spfli AS p ON p~carrid   = c~carrid
                              AND p~cityfrom = p_cityfr
                              AND p~cityto   = p_cityto )
         INNER JOIN sflight AS f ON f~carrid = p~carrid
                                AND f~connid = p~connid )
       INTO CORRESPONDING FIELDS OF TABLE itab.
```

Listing 5.29 Inner Join Example

Listing 5.30 shows an example using an outer join.

```
PARAMETERS p_cityfr TYPE spfli-cityfrom.

TYPES: BEGIN OF wa,
         carrid   TYPE scarr-carrid,
         carrname TYPE scarr-carrname,
         connid   TYPE spfli-connid,
      END OF wa.
DATA  itab TYPE SORTED ABLE OF wa
               WITH NON-UNIQUE KEY carrid.
START-OF-SELECTION.
SELECT s~carrid s~carrname p~connid
       FROM scarr AS s
       LEFT OUTER JOIN spfli AS p ON s~carrid   =  p~carrid
                                 AND p~cityfrom = p_cityfr
       INTO CORRESPONDING FIELDS OF TABLE itab.
```

Listing 5.30 Outer Join Example

There are certain restrictions for outer joins prior to SAP NetWeaver 7.40 SP5:

- To the right of the join operator, you can have only a table or a view. You can't have another join expression.

- With the ON condition, you can have only AND as the logical operator.
- Every comparison in the ON condition must contain a field from the table on the right.
- You can't have fields from the table on the right in the WHERE condition of the left outer join.

The new Open SQL post SAP NetWeaver 7.40 SP5 removes the above restrictions and also supports RIGHT OUTER JOIN. The RIGHT OUTER JOIN is similar to the LEFT OUTER JOIN, but with switched roles for the LEFT and the RIGHT tables. Additionally, JOIN expressions involving more than two tables have also been enhanced. Previously, the JOIN had to be performed left-to-right; that is, the left-most tables always had to be joined first (left-bracketing). This limitation no longer applies with new Open SQL.

Some of the new functionality is as follows:

- The previous restriction of having the fields of the RIGHT table for the ON condition is dropped.
- Apart from the EQUAL (=) operator, you can also have BETWEEN or GREATER THAN/LESS THAN operators in the ON condition
- For a LEFT OUTER JOIN statement, you can also use the fields of the RIGHT table in the WHERE clause.
- Implicit client handling is now supported, so you no longer have to compare the MANDT field for client-dependent tables.
- The maximum number of tables that you can use in a JOIN expression has been increased to 50.

Refer to the ABAP documentation to learn more about implementing JOIN functions. Using joins may seem a bit overwhelming initially, so we suggest you practice using FOR ALL ENTRIES and use joins only if you need data from many tables.

5.4 Processing Data from Databases via Internal Tables and Structures

Internal tables and structures are primarily used to process data from database tables. Because all transaction data is stored in database tables, we use internal tables and structures in almost all ABAP programs. Working with internal tables may seem a bit tricky initially, especially when working with multiple tables.

When processing data from multiple internal tables, you should always avoid nested loops (a loop within a loop) because they degrade performance. We always loop the main table and read the secondary tables.

When you have a requirement to process data from more than one table, you may struggle to identify which internal table to loop and which one to read. For example, you might have a requirement to process material and plant data, and this data is available in database table MARC. To print the plant details of a material in the output, write the code shown in Listing 5.31.

```
TYPES: BEGIN OF ty_marc,
         matnr TYPE matnr,
         werks TYPE werks_d,
       END OF ty_marc.
DATA : it_marc TYPE STANDARD TABLE OF ty_marc,
       wa_marc TYPE ty_marc.
PARAMETERS p_matnr TYPE matnr.
SELECT matnr werks from marc INTO TABLE it_marc WHERE matnr EQ p_matnr.
LOOP AT it_marc INTO wa_marc.
WRITE: / wa_marc-matnr, wa_marc-werks.
ENDLOOP.
```

Listing 5.31 Sample Code to Print Material Plant Data

Now, if the code in Listing 5.31 has to be expanded to print the plant description as the third column for every plant number, you need to select the plant description from another table, T001W, and compare against the plant data in table MARC, as shown in Listing 5.32.

```
TYPES: BEGIN OF ty_marc,
         matnr TYPE matnr,
         werks TYPE werks_d,
       END OF ty_marc.
TYPES : BEGIN OF ty_t001w,
         Werks TYPE werks_d,
         Name1 TYPE name1,
        END OF ty_t001w.
DATA : it_marc TYPE STANDARD TABLE OF ty_marc,
       wa_marc TYPE ty_marc,
       it_t001w TYPE STANDARD TABLE OF ty_t001w,
        wa_t001w TYPE ty_t001w.
```

```
PARAMETERS p_matnr TYPE matnr.
SELECT matnr werks from marc INTO TABLE it_marc WHERE matnr EQ p_matnr.
IF sy-subrc IS INITIAL.
SELECT werks name1 FROM t001w INTO TABLE it_t001w FOR ALL ENTRIES IN it_marc
WHERE werks EQ it_marc-werks.
ENDIF.
LOOP AT it_marc INTO wa_marc.
READ TABLE it_t001w INTO wa_t001w WITH KEY werks = wa_marc-werks.
WRITE: / wa_marc-matnr, wa_marc-werks, wa_t001w-name1.
ENDLOOP.
```

Listing 5.32 Printing Plant Descriptions

As you can see in Listing 5.32, we selected the required information from two database tables in two internal tables by matching the entries of the first table with the second table. After selecting the data in the respective internal tables, we loop one table and read the other table in the loop to print the output.

Now, how do you decide which table to loop and which table to read? This is where many beginners become confused, but the answer is simple: loop the table that has the main data, and read the table that has the secondary data. In this example, the main data we're printing is the plant information of the material; the plant descriptions are just secondary information. Therefore, we're looping the main table `MARA` and reading the secondary table `T001W`.

5.5 Introduction to the Debugger

We'll discuss debugging details in Chapter 17, but, for now, we'll cover the basics of debugging so that you can check your program execution to understand the program flow. The debugger is a useful tool not only to troubleshoot logical errors in your code but also to understand the program flow, and it can be an invaluable tool to understand your code better.

Two types of debugger are available: the Classic Debugger and the New Debugger. The New Debugger provides a lot more options than the Classic Debugger. If you're a beginner, we suggest that you use the Classic Debugger if you're using a pre-SAP NetWeaver 7.5 system; as you progress through this book, you should feel at home exploring the New Debugger.

With the release of SAP NetWeaver 7.5, the Classic Debugger has been declared obsolete, and it's no longer possible to set the Classic Debugger as your default debugger. You can still use the Classic Debugger by clicking the information icon in the debugger settings, however. Alternately, you can refer to Chapter 17 for a quick introduction to the New Debugger.

To always use the Classic Debugger, you need to make it the default debugger in your personal settings, as shown in Figure 5.14.

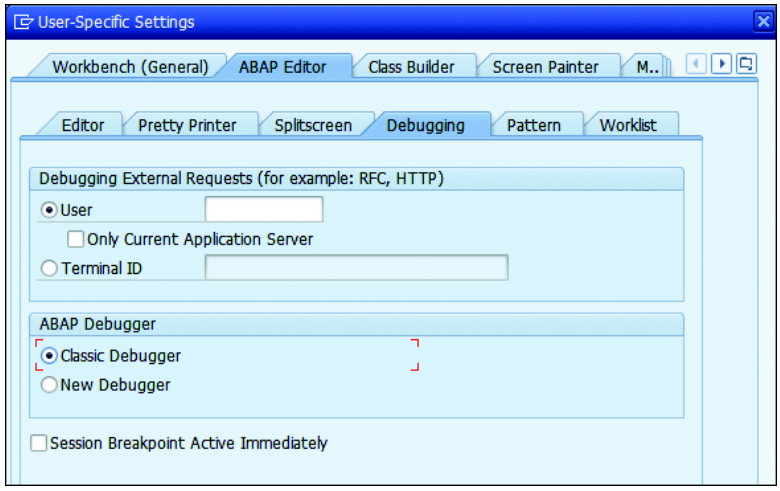


Figure 5.14 Setting Classic Debugger as Default

Open the **User-Specific Settings** dialog box by choosing **Utilities • Settings** from any ABAP Workbench tool, and then select **Classic Debugger** from the **Debugging** tab under the **ABAP Editor** tab. From now on, the system will always run the Classic Debugger.

To turn on debugging, you need to set a breakpoint in the program. A *breakpoint* interrupts program execution and loads the debugger when the program execution reaches the statement for which the breakpoint is set. You can set a breakpoint in a couple of ways, as shown in Figure 5.15:

❶ Click here to set or delete break point

❷ Double-click here to set or delete break point

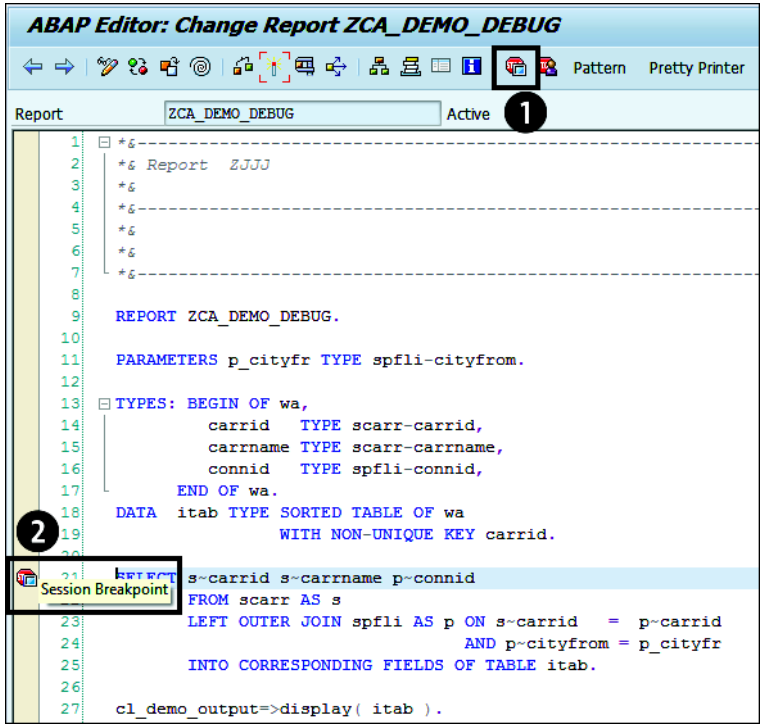


Figure 5.15 Setting Breakpoints

After you set a breakpoint, execute the program normally; the system will automatically hold the program execution and present the debugger screen when it reaches the breakpoint, as shown in Figure 5.16.

On the debugger screen, you can press F5 to execute one statement at a time or F8 to stop at the next breakpoint or to complete the program execution if there are no further breakpoints. If the debugger screen opens, and you want to exit the debugger, you can select **Debugging • Debugging Off** in the debugger screen (see Figure 5.17):

❶ Buttons to execute the code.

❷ Different views in debugger.

❸ Enter the data object/field name to check its contents. Double click on the field name to open detail view.

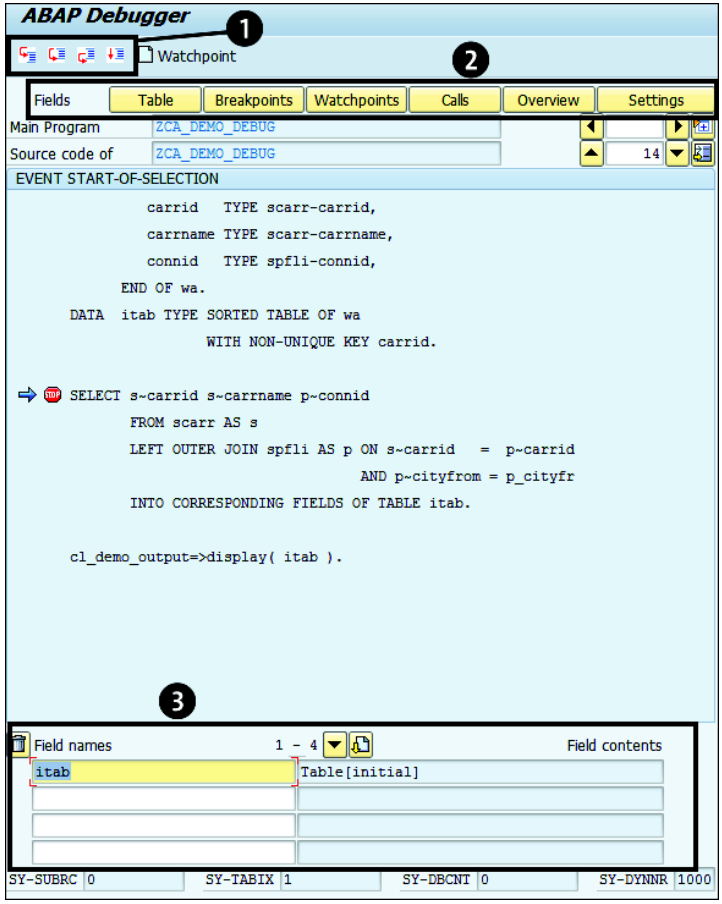


Figure 5.16 Debugger Screen

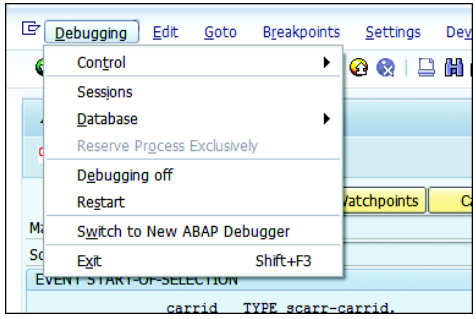


Figure 5.17 Menu Option to Exit Debugger

While you're in debug mode, you can type the name of a field to check its content. You can double-click on the field name to see a more detailed view of the field, as shown in Figure 5.18. This helps confirm that your variables are filled with the correct data and to see what happens after each statement is executed. This way, you don't have to imagine the program execution; you can see it step by step.

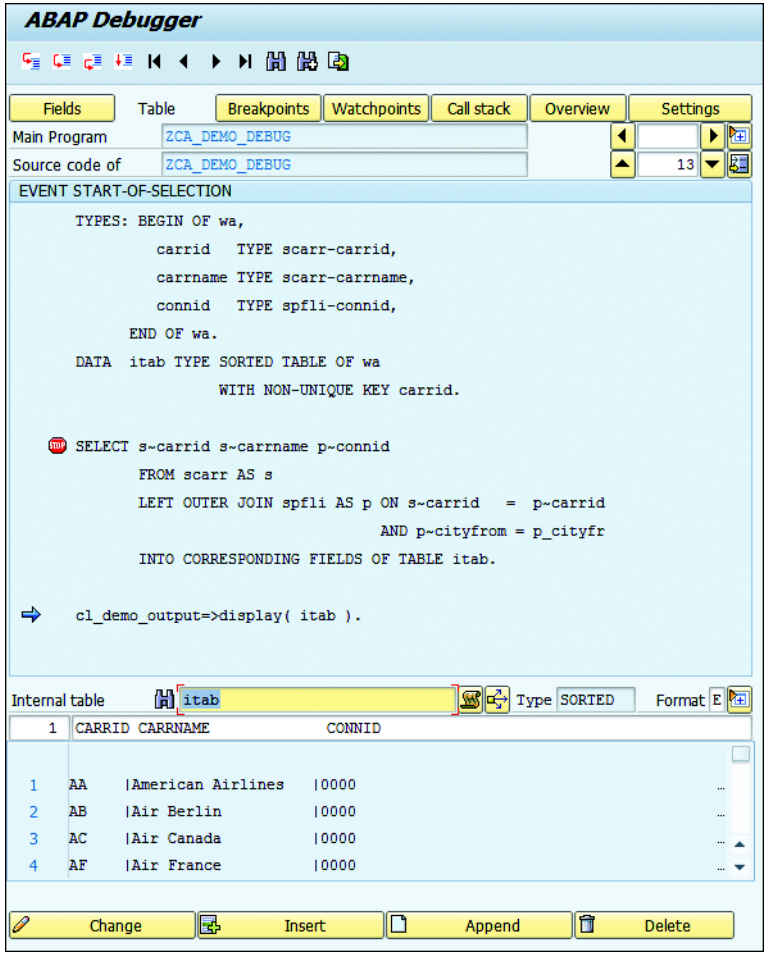


Figure 5.18 Detail View of a Field

We'll explore all the options in the Debugger later in Chapter 17; for now, you can use these basic options to better understand your program flow and troubleshoot any issues. For example, if you've written some code but don't see the output, you can

debug to see where the problem is, allowing you to check if the problem is with your selection logic or output logic.

5.6 Practice

We've explored quite a lot of concepts in this chapter, centering on processing data from database tables. The following task should help you put the theory you've learned into practice.

Ahead, you'll find a requirement to develop a simple report to display the header and item details of a billing document. Working on this report will require you to use the knowledge you've gained so far to define a simple selection screen for the report that will take the user input, fetch the relevant data from the database, and print it in the output:

Create a report to print the header and item details of a billing document. You can use Transaction VF03 as a reference. The report should take the billing document number as the input and print the header and item details of the document as output.

The header details of the billing document are stored in table `VBRK`, and line items are stored in table `VBRP`. Identify the fields in table `VBRK` that contain the header details, such as document number, billing date, payer, and net value, and print them as header information in the output.

Identify the fields in table `VBRP` that contain the item details, such as item number, description, billed quantity, unit of measure, net value, and tax amount. Print these details below the header details in the output with respective column headings.

5.7 Summary

In this chapter, we looked at structures and internal tables. We use structures and internal tables while working with any data that is in a structure format with rows and columns. We also explored different types of internal tables and discussed the key differences between them. We examined basic Open SQL statements that can be used to select data from database tables into your ABAP program for processing, and we explored the basics of debugging to help you debug any minor issues and understand the program flow.

In the next chapter, we'll discuss how to enable user interaction with your code.

Chapter 6
User Interaction

So far, we've covered the basic concepts for writing ABAP programs. In this chapter, we'll introduce the concepts that enable user interaction with programs.

In an SAP GUI environment, there are typically three types of screens: the selection screen, list screen, and general screen. In ABAP, a screen is also known as a *dynamic program* (*dynpro*). A dynpro is a component of a program and can be defined for three types of ABAP programs: *executable programs*, *module pool programs*, and *function groups*. A program can have multiple dynpros.

A dynpro consists of a screen, dynpro fields, and dynpro flow logic. General dynpros are created with the Screen Painter tool, and special dynpros (selection screens, list screens) are created implicitly using specific keywords. For example, `SELECT-OPTIONS` and `PARAMETERS` define selection screens, and `WRITE` defines a list screen.

Dynpro flow logic contains processing blocks for events that are triggered before a screen is displayed to the user and after user action on the displayed screen. Using these processing blocks, you can control the behavior of a dynpro and react to user actions. You'll learn about various processing blocks in Chapter 7. Dynpros, together with the new web-oriented features (e.g., ABAP Web Dynpro) of SAP NetWeaver AS ABAP, form the basis of user dialogs in an ABAP system.

In this chapter, we'll discuss the rudimentary concepts of selection screens to get you started with basic report development, and you'll use all the concepts discussed so far in this book to develop a report program (Section 6.1). This will allow you to understand the practical application of these concepts. We'll defer an in-depth discussion of selection screens to Chapter 14, and we'll discuss the other two screens in Chapter 12 (general screens) and Chapter 13 (list screens).

In Section 6.2, we'll discuss messages so that you can not only take user inputs for your program through the selection screen but also validate the input and show a message to the user accordingly.

6.1 Selection Screen Overview

Selection screens are special dynpros that can be defined in certain program types, for example, executable programs, function groups, and module pools. Selection screens are defined in the global declaration area of an ABAP program with the statements `SELECT-OPTIONS`, `SELECTION-SCREEN`, and `PARAMETERS`, without using the Screen Painter tool. The screens of the selection screens can contain a subset of the screen elements of general dynpros.

Selection screens are typically used in report programs, in which you first present the user with a screen to provide the selection criteria and then pull the required data from the database based on user input to be presented as the report output.

For example, if we want to develop a report to show a flight's status, we first need to take the input (selection criteria) from the user to determine the flight number. The user typically provides this input on a selection screen. After processing, we show the result on a different screen (either list dynpro or general dynpro).

Selection screens essentially have two tasks:

- Enables users to provide selection criteria for a report program
- Provides an interface for data transfer between two executable programs when a program is called internally using the `SUBMIT` statement

Each screen is assigned a four-digit number. A program can have a *standard selection screen* and a *user-defined selection screen*. Screen number 1000 is assigned to the standard selection screen by default. The standard selection screen is created automatically when you use the `PARAMETERS` or `SELECT-OPTIONS` keyword in the global declaration area of the program. Other user-defined selection screens can be created for the program using the `SELECTION-SCREEN BEGIN OF SCREEN` keyword with each user-defined screen assigned a unique screen number.

The `SELECTION-SCREEN` statement can also be used to define different blocks with frames to better organize the selection screen fields (see Figure 6.1). All screens (e.g., general screens, selections screens, and list screens) share the same namespace in the program, so you can't assign the same screen number to different types of screens in the same program.

When you use the `PARAMETERS` or `SELECT-OPTIONS` keyword in the program and activate it, the system automatically creates the screen with all the screen elements and dynpro flow logic. If the `SELECTION-SCREEN` keyword isn't used to define a user-defined

screen, then the program automatically creates a standard selection screen with dynpro number 1000 and with all the PARAMETERS and SELECT-OPTIONS fields belonging to this standard screen.

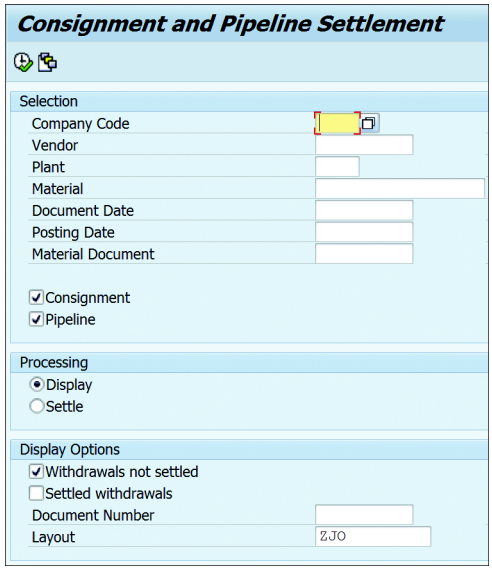


Figure 6.1 Selection Screen Fields Organized with Blocked Frames

You can define various screen elements on the selection screen, such as input fields, checkboxes, radio buttons, list boxes, and push buttons. All the screen elements are defined in the program using the PARAMETERS keyword. The SELECT-OPTIONS keyword defines a range field on the selection screen, which supports more complex selection criteria.

In the following subsections, we'll discuss the PARAMETERS, SELECT-OPTIONS, and SELECTION-SCREEN keywords in more detail.

6.1.1 PARAMETERS

Parameters are components of a selection screen that are assigned a global elementary data object in the ABAP program and an input field on the selection screen. In other words, the PARAMETERS keyword performs two tasks:

- Defines a data object in the program
- Defines a screen element (e.g., an input field or checkbox) on the selection screen

The system automatically links a program data object with the screen field to facilitate automatic data transfer between the screen and the program. Thereby the user inputs on the screen are automatically transferred to the ABAP program and stored in the corresponding data object, which can then be accessed by ABAP statements.

The syntax for using the PARAMETERS keyword is somewhat similar to that of the DATA keyword, with its own set of additions. The PARAMETERS field is typed by referring to a data type from which it derives its technical attributes. Listing 6.1 shows the syntax to use the PARAMETERS statement.

```
PARAMETERS {para TYPE data_type}
           [type_options]
           [screen_options]
           [value_options]
```

Listing 6.1 PARAMETERS Keyword Syntax

Listing 6.2 shows an example of the PARAMETERS keyword in action.

```
PARAMETERS p_name TYPE c LENGTH 20.
PARAMETERS p_name(20) TYPE c.
PARAMETERS p_carrid TYPE s_carr_id.
PARAMETERS p_chk AS CHECKBOX DEFAULT 'X'.
```

Listing 6.2 PARAMETERS Keyword Example

The name of the parameter field para (in the syntax shown in Listing 6.1) can contain a maximum of eight characters. This statement is allowed in the global declaration part of the program.

Similar to the DATA statement, the length can only be specified in the PARAMETERS statement if the data type specified in TYPE_OPTIONS has a generic length (types c, n, p, and x).

The PARAMETERS statement has the following effects:

- Declares a global variable of the specified length and type in the program.
- Creates an input field with the same name and corresponding type in the new row of the current selection screen. The length of the input field on the screen is determined from the parameter definition. The maximum visible length of the input field is 45, so if the parameter length is more than 45, the field will be displayed

with movable content. The visible length of the field can be controlled using the VISIBLE LENGTH addition with the PARAMETERS keyword.

- Automatically creates a label for each field to the left side of the input field. The maximum length of the field label is 30, and it displays either the name of the parameter or the selection text to which the parameter is assigned in the text elements (Section 6.1.4) of the program.
- Allows for the attributes of the screen elements to be modified by passing SCREEN_OPTIONS during the declaration of the parameters.

In the following subsections, we'll discuss the type options, screen options, and value options that can be supplied with the PARAMETERS statement.

TYPE_OPTIONS

The PARAMETERS keyword can be supplied with different type options. These options define the data type of the parameter. If no type options are specified, the default option is TYPE c with LENGTH 1. The data type can be defined by referring to an existing data type of the program, to a data object, or to a data type from the ABAP Data Dictionary.

If the PARAMETERS field is referred to a data type from the ABAP Data Dictionary, it adopts all the screen-relevant properties (e.g., field labels or input helps) defined for the data type in the ABAP Data Dictionary. If a domain is attached to the data element, conversion routines defined in the domain will be executed to convert the data to internal and external formats during the data transport from and to the input field, respectively.

There are three ways to provide TYPE_OPTIONS:

- TYPE data_type [DECIMALS dec]
- LIKE dobj
- LIKE (name)

The following subsections look at each of these options.

TYPE data_type [DECIMALS dec]

The TYPE data_type [DECIMALS dec] addition types the parameter with data_type, where data_type may be any of the following:

- A predefined ABAP type, with the exception of f and xstring
- A nongeneric data type from the ABAP Data Dictionary

- A nongeneric public data type of a global class
- A user-defined elementary data type from the same program (which isn't of the type `f` or `xstring`), already defined using the `TYPES` keyword

If the predefined `TYPE p` (packed decimal) is specified, the `DECIMALS` addition can be used to define the number of decimal places.

If the parameter is referred to a data element to which a domain of `TYPE CHAR`, `LENGTH 1`, and the fixed values `"X"` and a space is attached, the input field on the selection screen is automatically displayed as a checkbox.

The following is an example of an elementary data type:

```
PARAMETERS p_carrid TYPE c LENGTH 3.
```

The next example automatically inherits the F1 and F4 help from the ABAP Data Dictionary and refers to the data element:

```
PARAMETERS p_carrid TYPE s_carr_id.
```

The following example is a declaration of a parameter with reference to the component `carrid` of the database table `spfli`:

```
PARAMETERS p_carrid TYPE spfli-carrid.
```

LIKE dobj

Using the `LIKE dobj` addition, the parameter adopts all the properties of a data object `dobj` already declared in the program.

Here's an example of this addition in use:

```
DATA v_carrid TYPE s_carr_id.
PARAMETERS p_carrid LIKE v_carrid.
```

TYPE versus LIKE

`TYPE` is used to refer to an elementary data type, local data type in a program, or data type in the ABAP Data Dictionary. `LIKE` is used to refer to another data object in the program.

To support backward compatibility, it's still possible to use `LIKE` to refer to an ABAP Data Dictionary table or flat structure; however, this usage is obsolete and should be avoided.

LIKE (name)

The LIKE (name) addition allows you to supply the field reference dynamically. name expects a flat character data object containing the name of a component in a flat structure from the ABAP Data Dictionary in uppercase. The parameter is created internally with data TYPE c and LENGTH 132, but it's displayed on the selection screen with a length, field label, input help, and field help per the data type specified in name.

Listing 6.3 shows an example of dynamically assigning the data type for the input field. In this example, we're passing the field name SCOL_FLIGHT_MEAL-CARRID at runtime. The statement comp = 'SCOL_FLIGHT_MEAL-CARRID' in the example is executed before the selection screen is called because it's maintained under the AT SELECTION-SCREEN OUTPUT event. The ABAP runtime environment triggers various selection screen events that we can maintain in the program to process selection screens.

You'll learn about various selection screen events in detail in Chapter 14.

```
DATA comp TYPE c LENGTH 30.
PARAMETERS p_dyn LIKE (comp).
AT SELECTION-SCREEN OUTPUT.
  comp = 'SCOL_FLIGHT_MEAL-CARRID'.
```

Listing 6.3 Dynamically Assigning Data Type for Input Field

In Listing 6.3, the keyword AT SELECTION-SCREEN OUTPUT defines an event that ensures the code under it is executed before the selection screen is called. Because this event is triggered before showing the selection screen to the user, we can make use of it to assign the properties dynamically to the selection screen field.

We'll explore all the program events in Chapter 7.

SCREEN_OPTIONS

These options allow you to manipulate screen fields, including declaring the input field as a required field or hiding the input field on the selection screen. Using SCREEN_OPTIONS, the input field can be displayed as a checkbox, radio button, or dropdown list box.

The following subsections look at the different additions for SCREEN_OPTIONS.

OBLIGATORY

The OBLIGATORY addition defines the input field on the selection screen as a required (obligatory) field. The program can't be executed if the obligatory field is empty. The

runtime system automatically performs a validation to check if all obligatory fields are filled with input values when the user selects the **Execute** function F8 to exit the selection screen. If the obligatory field isn't filled, the user can only use the functions **Back**, **Exit**, or **Cancel** to leave the selection screen.

An example of this addition is as follows:

```
PARAMETERS p_carrid TYPE s_carr_id OBLIGATORY.
```

NO-DISPLAY

When the `NO-DISPLAY` addition is used, no screen elements are generated for the parameter on the selection screen. This is useful when you want to use the parameter only as part of the interface defined by the selection screen in an executable program that can be supplied with a value by the calling program with the `SUBMIT` statement. The `SUBMIT` statement is used to execute an executable program internally from another program.

An example of this addition is as follows:

```
PARAMETERS p_carrid TYPE s_carr_id NO-DISPLAY.
```

VISIBLE LENGTH vlen

The `VISIBLE LENGTH vlen` addition controls the visible length of a screen field. Here, `vlen` is entered directly as a positive number. If `vlen` is less than the length of the parameter and less than the maximum visible length (45), then the input field is displayed per the length defined in `vlen`, with movable content. Otherwise, the addition is ignored. An example of this addition is as follows:

```
PARAMETERS: p_carrid TYPE s_carr_id VISIBLE LENGTH 2.
```

AS CHECKBOX

The `AS CHECKBOX` addition generates the input field as a checkbox with the corresponding description next to it on the right. The value of the field is filled with `'X'` or a space when the checkbox is selected or deselected, respectively. The checkbox can be selected by default by assigning a default value `'X'` during the declaration. This parameter must be created with `TYPE c` and `LENGTH 1`. Checkboxes are used to allow multiple selections.

An example of this addition is as follows:

```
PARAMETERS p_chk AS CHECKBOX.
PARAMETERS p_chk AS CHECKBOX DEFAULT 'X'.
```

RADIOBUTTON GROUP

The RADIOBUTTON GROUP addition generates the input field as a radio button with the corresponding description next to it on the right. The radio button is selected if the value of the field is X; otherwise, it isn't selected.

Radio buttons are always grouped together because they enable the user to select from several options. Only one of the radio buttons can be selected at once in a group. *Groups* define the radio button groups for the parameter. Two or more radio buttons are assigned to the same group by entering the name of the group directly as a character string with a maximum of four characters. Within a selection screen, there must be a minimum of two parameters in the same radio button group. There can't be more than one radio button group with the same name in one program.

In a radio button group, only one parameter can be selected by default, which means that the DEFAULT addition can be used with only one parameter of the group. By default, the first parameter in a radio button group is set to the value X, and the rest are set to space. This parameter must be created with TYPE c and LENGTH 1.

An example of the radio button group is shown in Listing 6.4.

```
PARAMETERS: p_day RADIOBUTTON GROUP rb1,
            p_month RADIOBUTTON GROUP rb1 DEFAULT 'X'.

PARAMETERS: p_hour RADIOBUTTON GROUP rb2,
            p_sec RADIOBUTTON GROUP rb2.
```

Listing 6.4 Radio Button Group

AS LISTBOX VISIBLE LENGTH vlen

The AS LISTBOX VISIBLE LENGTH vlen addition generates a dropdown list box for an input field on the selection screen. The list box is automatically filled with values if the parameter is created with a data type from the ABAP Data Dictionary, and the data type is associated with the input help in the ABAP Data Dictionary. Otherwise, a single-row list box is displayed containing the current value of the parameter.

You must use the VISIBLE LENGTH addition to specify the visible length of the input field. An example of this addition is shown in Listing 6.5.

```
PARAMETERS p_carrid TYPE spfli-carrid
                    AS LISTBOX VISIBLE LENGTH 20
                    DEFAULT 'LH'.
```

Listing 6.5 AS LISTBOX VISIBLE LENGTH vlen

Figure 6.2 shows different screen elements on the selection screen defined using the PARAMETERS keyword.

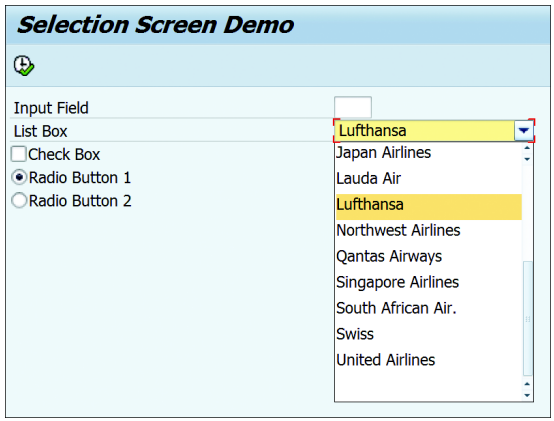


Figure 6.2 Different Screen Elements on Selection Screen

VALUE_OPTIONS

Using VALUE_OPTIONS, it's possible to define a default value or allow lowercase letters to be transported to the data object. It's also possible to set a search help, an SPA/GPA parameter to be bound, or a check to be executed against a value list. For simplicity, we'll defer the discussion of these options to Chapter 14.

6.1.2 SELECT-OPTIONS

The PARAMETERS statement defines a single input field in which the user can input a single value. For example, say that a user wants to run a report to see information about a billing document. With PARAMETERS, the user can run the report to check the details of one document at a time because the parameters field can only take one document number as input.

However, SELECT-OPTIONS comes into play when the user wants to check the details of multiple documents at once. SELECT-OPTIONS defines an internal range table for a data object.

To better understand how SELECT-OPTIONS works, you need to first understand the range table, and then we'll discuss multiple selection windows.

Range Table

A *range table* is an internal table with a special structure, the content of which can be analyzed with the relational operator IN. For example, to check if a particular material number falls within a given range of materials, we can use the following statement:

```
IF v_matnr IN rt_matnr.

ENDIF.
```

In this example, v_matnr contains the material number we want to check, and rt_matnr is a special range table that contains the range of materials. The IN operator automatically scans the range table to check if the value falls within the range.

A range table can be defined in the program via the TYPE RANGE OF addition with the DATA keyword, for example:

```
DATA rt_carrid TYPE RANGE OF s_carr_id.
```

A range table has four fields (columns): SIGN, OPTION, LOW, and HIGH. Figure 6.3 shows the components of a range table as seen in debug mode; note the **Type** and length (**Lngth**) of each field.

No.	Component name	Type	Lngth	Contents
1	SIGN	C	1	
2	OPTION	C	2	
3	LOW	C	3	
4	HIGH	C	3	

Figure 6.3 Components of a Range Table

The four range table components can be described as follows:

- **SIGN**
  This field is of TYPE c and LENGTH 1. The content of SIGN determines for every row whether the values in the LOW and HIGH fields are included or excluded. Valid values are I for include and E for exclude.
- **OPTION**
  This field is of TYPE c and LENGTH 2. OPTION contains the relational operator for the selection condition of the row. Valid operators are EQ (equal to), NE (not equal to), GE (greater than or equal to), GT (greater than), LE (less than or equal to), LT (less than), CP (matches pattern), and NP (does not match pattern) if column HIGH is initial. If column HIGH isn't initial, BT (between) and NB (not between) are valid. For the options CP and NP, the data type of the columns LOW and HIGH must be c.

- **LOW**
  The length and type of the LOW field depends on the data type reference provided while declaring the range table. In Figure 6.3, the range table is defined with reference to s_carr_id, so it's of TYPE c and LENGTH 3. This column is designated for the comparison value or the lower interval limitation.
- **HIGH**
  Similar to the LOW field, the LENGTH and TYPE of the HIGH field depends on the data type reference provided while declaring the range table. This column is designated for the upper interval limitation.

The system automatically analyzes the row of a range table when used with the IN operator. IN operators are very useful when a complex comparison of values is required.

When we use SELECT-OPTIONS, the system defines an internal range table with a header line in the program and creates two input fields, low and high, on the current selection screen.

To define a SELECT-OPTIONS field, use the FOR addition with the SELECT-OPTIONS keyword. The system expects a data object after FOR, for example:

```
DATA v_carrid TYPE s_carr_id.
SELECT-OPTIONS s_carrid FOR v_carrid.
```

This code creates a range table with a header line for which the LOW and HIGH fields are of type v_carrid in the program. It also creates two input fields on the selection screen, s_carrid-low ❶ and s_carrid-high ❷, with a **Multiple Selection** button ❸, as shown in Figure 6.4.

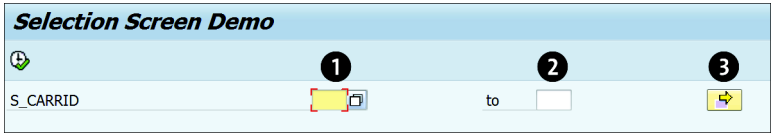


Figure 6.4 Range Field on the Selection Screen

Multiple Selection Window

With SELECT-OPTIONS, the user can provide a value range or use the **Multiple Selection** button to enter a more complex selection criterion. Figure 6.5 shows the **Multiple Selection** dialog box that's presented upon clicking the **Multiple Selection** button:

❶ Selection tabs

❷ Enter values here

❸ Operator selection

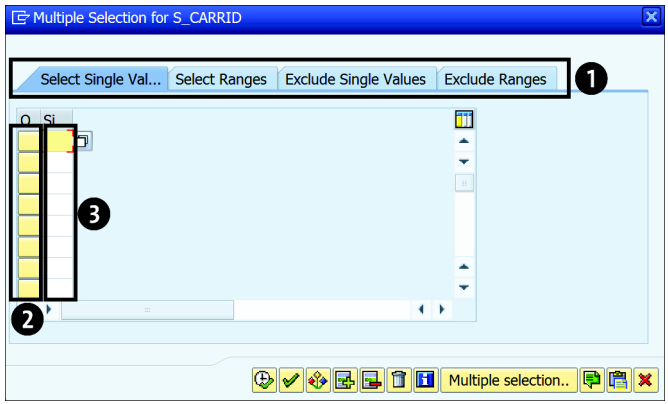


Figure 6.5 Multiple Selection Dialog Box

The **Multiple Selection** dialog box includes four tabs to input selection criteria:

- **Select Single Values**
  The **Select Single Values** tab allows the user to input multiple single values to be considered for selection. The user can also select the options for each value by using the **Options** button, as shown in Figure 6.6.

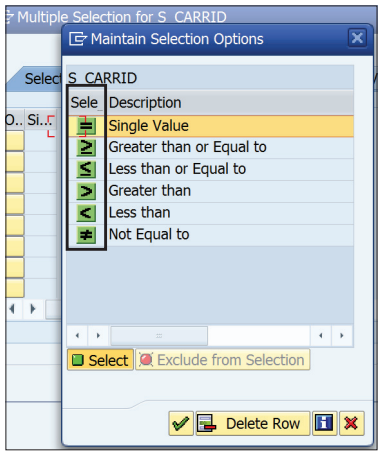


Figure 6.6 Options Button for Single Values

The entries made in this tab will be stored in the range table with **SIGN** as `I` (include), **OPTION** as the operator selected (default is `EQ`), and **LOW** field with the input value. The **HIGH** field for this row will be empty.

Figure 6.7 shows the rows of the `SELECT-OPTIONS` range table as seen in debug mode for the corresponding entries made in the **Select Single Values** tab. Here you can see:

❶ Entries made on the selection screen

❷ Entries stored in the range table

When used with the `IN` operator, the system automatically processes the row per the values in the **SIGN**, **OPTION**, **LOW**, and **HIGH** fields.

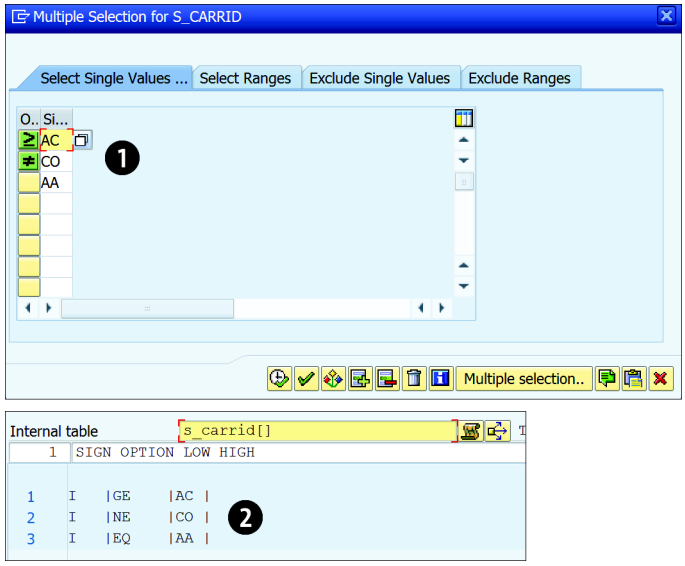


Figure 6.7 Range Table in Debug Mode for Multiple Selection Dialog Box Entries

- **Select Ranges**

  This tab allows you to enter values in a range that should be included in the search. The operator allows you to specify whether the values are inside the range or outside the range. Figure 6.8 shows the options available for ranges.

  The **SIGN** field of the range table will be `I`, **OPTIONS** will either be `BT` or `NB` based on the options selected for the row, the **LOW** field will be filled with the low value, and the **HIGH** field will be filled with the high value entered on the screen.

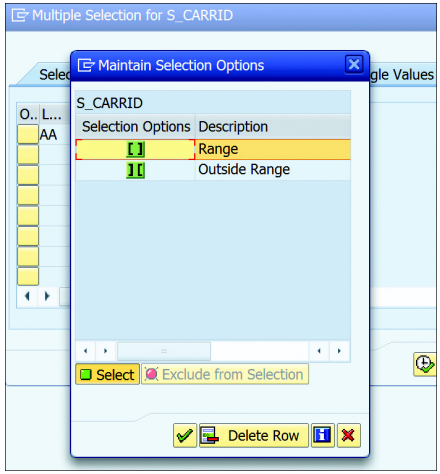


Figure 6.8 Operator for Ranges

The high value entered on the screen should be greater than the low value; otherwise, the screen will throw an error message. Figure 6.9 shows the corresponding entries in the range table for the values entered on the screen:

❶ Entries made on the selection screen

❷ Entries stored in the range table

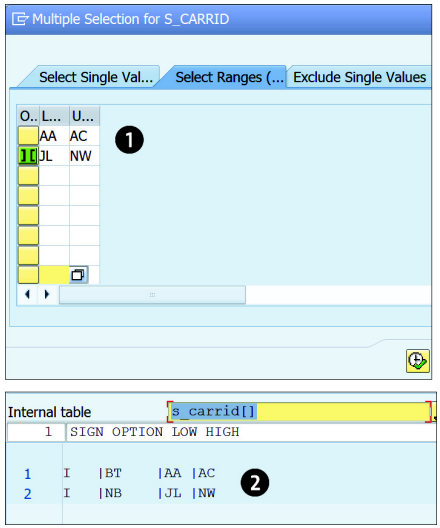


Figure 6.9 Range Table Values per Select Ranges Tab

- **Exclude Single Values**
  The entries in this tab are similar to those of the **Select Single Values** tab, with the exception that the values maintained in this tab are excluded from selection. In other words, the **SIGN** field of the row will be filled with `E` (exclude).
- **Exclude Ranges**
  The entries in this tab are similar to those of the **Select Ranges** tab, with the exception that the values maintained in this tab are excluded from selection.

The **SIGN** field for rows in **Select Values** (both single values and ranges) will be filled with `I`, and **Exclude Values** (both single values and ranges) will be filled with `E`.

Let's consider a requirement in which the user wants to run a report to process billing documents. The user's selection criteria are as follows:

1. Select document numbers **100**, **122**, **140**.
2. Select all the documents from **200** to **400** and from **500** to **600**.
3. Exclude document numbers **222** and **320** from the previous range.
4. Exclude document numbers **520** to **540**.

For the example selection criteria, the user will make the entries in the multiple selection window in different tabs, as shown in Figure 6.10.

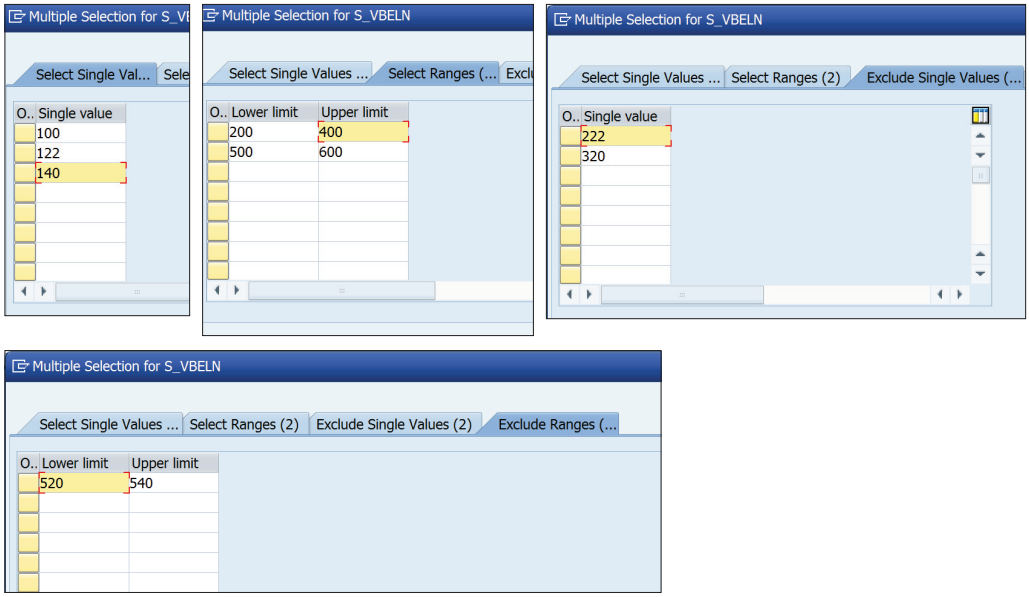
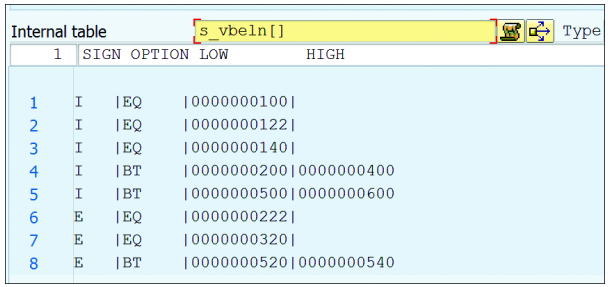


Figure 6.10 Entries in Different Tabs of Multiple Selection Window

The entries in Figure 6.10 will be transferred to the range table, as shown in Figure 6.11. Notice the value in the **SIGN** field for the entries made in the **Select** and **Exclude** tabs.


Internal table s_vbeln[] Type

	SIGN	OPTION	LOW	HIGH
1	I	EQ	0000000100	
2	I	EQ	0000000122	
3	I	EQ	0000000140	
4	I	BT	0000000200	0000000400
5	I	BT	0000000500	0000000600
6	E	EQ	0000000222	
7	E	EQ	0000000320	
8	E	BT	0000000520	0000000540

Figure 6.11 Corresponding Entries in a Range Table for Selection Criteria in the Selection Screen Demo Dialog

For the entries made in the **Select** tabs of the **Multiple Selection** window, the **SIGN** field is filled with the value `I`, and for the entries made in the exclude tabs of the **Multiple Selection** window, the **SIGN** field is filled with the value `E`. The values in the **LOW** and **HIGH** fields are automatically filled with leading zeros with the help of a conversion routine attached to the domain.

The code in Listing 6.6 shows the usage of `SELECT-OPTIONS` to process these multiple documents. In this code, we're processing the basic header data of the documents provided in the selection criteria. The code highlights the declaration of `SELECT-OPTIONS` and its subsequent use in the `SELECT` query to filter the records per the user selection criteria using the `IN` operator.

```
TYPES : BEGIN OF ty_vbrk,
        VBELN TYPE VBELN_VF, " Document Number
        FKART TYPE FKART,    " Billing Type
        FKDAT TYPE FKDAT,    " Billing Date
        NETWR TYPE NETWR,    " Net Value
        KUNRG TYPE KUNRG,    " Payer
  END OF ty_vbrk.
DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
       wa_vbrk TYPE ty_vbrk.
SELECT-OPTIONS s_vbeln FOR wa_vbrk-vbeln.
SELECT   VBELN
         FKART
```

```
          FKDAT
          NETWR
          KUNRG
               FROM VBRK
               INTO TABLE it_vbrk
               WHERE vbeln IN s_vbeln.
LOOP AT it_vbrk INTO wa_vbrk.
WRITE:/ 'DOCUMENT NUMBER:' , wa_vbrk-vbeln,
      /  'Billing Type :' , wa_vbrk-fkart.
      / 'PAYER:' ,  wa_vbrk-kunrg,
      /  'BILLING DATE :', wa_vbrk-fkdat,
      /  'NET VALUE :' , wa_vbrk-netwr LEFT-JUSTIFIED.

ENDLOOP.
```

Listing 6.6 Sample Code to Process Multiple Billing Documents

6.1.3 SELECTION-SCREEN

The `SELECTION-SCREEN` keyword has several uses. It creates standalone selection screens and selection screen layouts, and it uses elements from other selection screens. We'll provide some examples of using this keyword to define selection screen layouts in this section. A more in-depth discussion of the `SELECTION-SCREEN` keyword can be found in Chapter 14.

The `SELECTION-SCREEN` keyword can be supplied with various layout options to influence the screen layout:

- **`SKIP`**
  Creates a blank line.
  Example: `SELECTION-SCREEN SKIP.`
- **`ULINE`**
  Creates a horizontal line.
  Example: `SELECTION-SCREEN ULINE.`
- **`BEGIN OF BLOCK`**
  Allows you to organize the selection screen elements into multiple blocks. The block is closed with the `END OF BLOCK` statement, and it supports defining frames and titles for the frames. Arranging the screen elements into multiple blocks is very useful on selection screens with many fields, as shown in Listing 6.7.

```
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME
                                        TITLE text-001.
PARAMETERS : p_carrid TYPE s_carr_id,
             p_connid TYPE s_conn_id.
SELECTION-SCREEN END OF BLOCK b1.

SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME
                                        TITLE text-002.
PARAMETERS : p_alv RADIOBUTTON GROUP rb1,
             p_list RADIOBUTTON GROUP rb1.
SELECTION-SCREEN END OF BLOCK b2.
```

Listing 6.7 Organizing Selection Screen Fields into Multiple Blocks

The code in Listing 6.7 generates the selection screen layout shown in Figure 6.12. The frame titles are maintained using text elements.

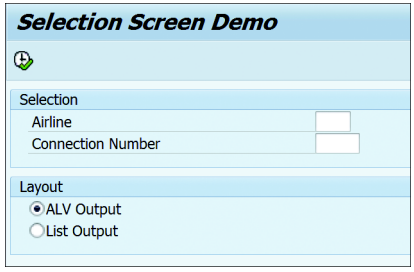


Figure 6.12 Selection Screen Layout Generated by Code in Listing 6.7

6.1.4 Selection Texts

You can maintain labels for selection screen fields via selection texts. Selection texts are maintained in the ABAP program by choosing **Goto • Text Elements • Selection Texts**.

Figure 6.13 shows the selection text maintenance screen.

Here, you can manually maintain the text—or pick it from the ABAP Data Dictionary—if the screen field is referred to a data element. To pick the text from the ABAP Data Dictionary, simply select the checkbox to the right of the corresponding field in the **Dictionary ref.** column, as shown in Figure 6.13. This checkbox is disabled if the field isn't referred to any ABAP Data Dictionary object.

Click the **Activate** icon in the application toolbar to apply the changes. The selection screen fields will show the label, as shown in Figure 6.13. The selection screen is generated when the program is activated. If any changes are made to the selection screen fields, the program should be activated to regenerate the selection screen and reflect the changes. The selection screen of the program should not be edited manually using the Screen Painter tool because any changes made manually will be lost when the selection screen is regenerated.

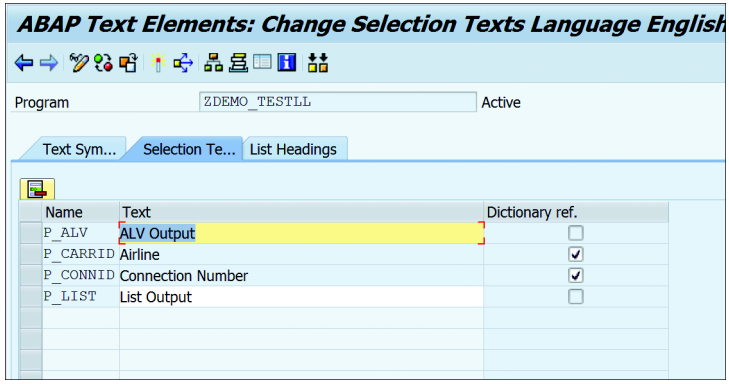


Figure 6.13 Selection Text Maintenance Screen

6.2 Messages

Messages allow you to communicate with users from your programs. They're used mainly when the user has made an invalid entry on a screen. Messages are raised in the program code via the `MESSAGE` statement. There are different types of messages that can be displayed based on requirements. The message text can be maintained locally in the program or globally using a message class.

An example of a `MESSAGE` statement in use is as follows:

```
MESSAGE 'The input is invalid' TYPE 'E'.
```

The addition `TYPE 'E'` in the statement specifies it as an error message.

In this section, we'll look at the various types of messages that can be displayed in an ABAP program. We'll also explore the different ways messages can be displayed and how messages can be translated to other languages.

6.2.1 Types of Messages

Message processing depends on the message type specified in the MESSAGE statement and the program context in which the statement occurs. The system behavior after a particular message type is sent is context-dependent. Table 6.1 lists all the possible types of messages that can be sent from ABAP programs. The type of message shown influences how the program behaves in a post-error situation and defines how the ABAP runtime should process the message. Always use the right message type for the condition, and keep the message informative enough for the user to understand the error.

You should always validate the user inputs and show specific and relevant messages to the user to identify any issues with input values.

Message Type	Meaning	Notes
A	Termination message	The message appears in a dialog box and the program terminates. When the user has confirmed the message, the control returns to the next-highest area menu. This type of message should be limited to situations in which the error can't be handled by the current task or when system-related errors occur.
E	Error message	Depending on the program context, an error dialog appears or the program terminates. Program execution won't continue unless the error is corrected. These types of messages are used for input validations or when program execution can't proceed due to an error.
I	Information message	The message appears in a dialog box. After the user has confirmed the message, the program continues immediately after the MESSAGE statement. This type of message is used to show informative messages, such as successful database updates.
S	Status message	The program continues normally after the MESSAGE statement, and the message is displayed in the status bar of the next screen. This type of message is used to display the status of any action.

Table 6.1 Types of Messages

Message Type	Meaning	Notes
W	Warning message	Depending on the program context, an error dialog appears or the program terminates. The user can press Enter to continue with program execution. This type of message is used to warn the user of missing information.
X	Exit message	No message is displayed, and the program terminates with a short dump. Program terminations with a short dump normally only occur when a runtime error occurs. Message type X allows you to force a program termination. The short dump contains the message ID.

Table 6.1 Types of Messages (Cont.)

The type of message is specified in the MESSAGE statement directly, for example:

```
MESSAGE 'The input was invalid' TYPE 'E'
MESSAGE 'Database update successful' TYPE 'I'
MESSAGE 'Key parameters are missing' TYPE 'W
```

6.2.2 Messages Using Text Symbols

One major disadvantage of using text literals to maintain a message in program code is that they can't be reused in the program or customized for the user logon language. By using text symbols, you can reuse the messages within the program and maintain translations wherein the translated text can be automatically picked up by the system based on a user's logon language.

To maintain text symbols, choose **Goto • Text Elements • Text Symbols** in the ABAP Editor, as shown in Figure 6.14.

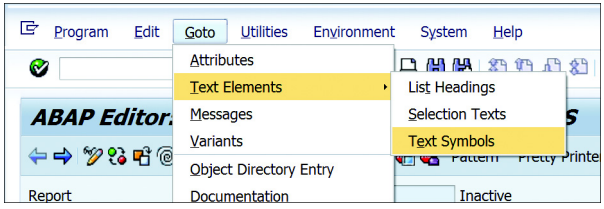


Figure 6.14 Menu Path to Maintain Text Symbols in an ABAP Program

On the **Change Text Symbols** screen (see Figure 6.15), you can assign a unique three-digit alphanumeric value to each text symbol that can then be called from the ABAP program.

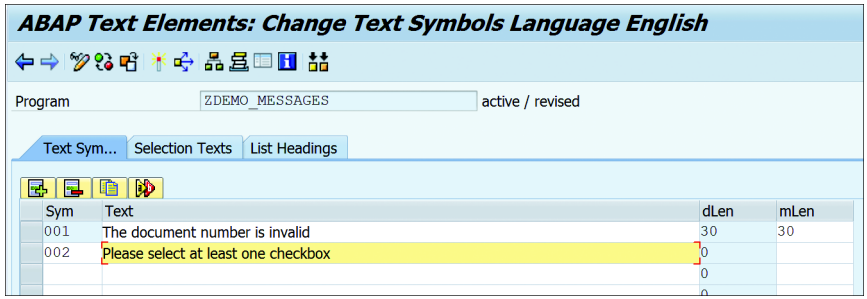


Figure 6.15 Change Text Symbols Screen

The text symbol editor also has the option to maintain translations for each text by choosing **Goto • Translation** from the **Change Text Symbols** screen, as shown in Figure 6.16.

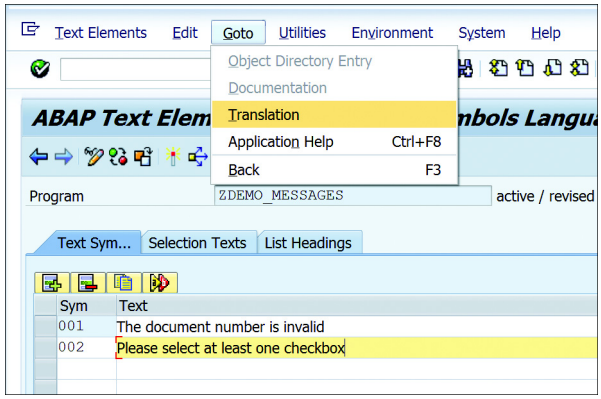


Figure 6.16 Menu Path to Access Translation Editor

To use a text symbol in a message, simply use `text-nnn` to call the text symbol. Here, `nnn` is the three-digit text symbol number, for example:

```
MESSAGE text-001 TYPE 'E'.
```

The system will automatically take care of pulling the respective translated texts to support multiple logon languages. If there's no translation maintained for the given logon language, a blank message will be shown.

6.2.3 Messages Using Message Classes

Text symbols maintained in the program are local, so they can only be reused within the program. However, what if you need to share message texts across multiple related programs? Maintaining the same message texts and their translations in multiple programs can be redundant and a waste of time because any changes to texts should also be manually maintained in each program. To maintain message texts globally, use a *message class*.

Global message texts are stored in table T100 and can be configured using Transaction SE91. When creating a new message class, it should be in a customer namespace, starting with Y or Z.

To create a new message class, open Transaction SE91. On the **Message Maintenance** initial screen, enter a unique name for your message class, and click the **Create** button, as shown in Figure 6.17.

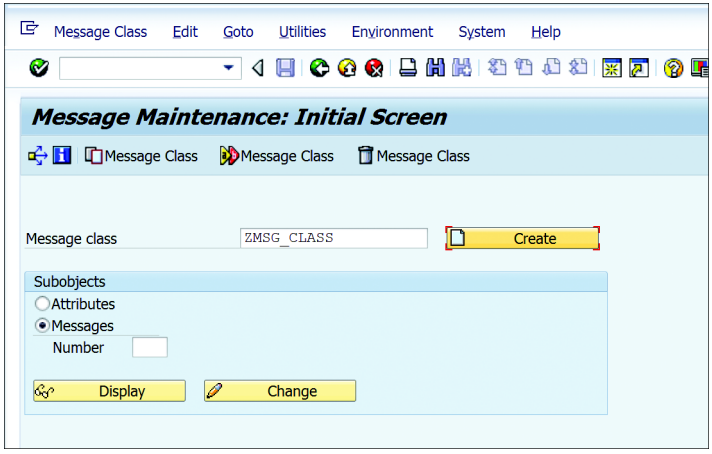


Figure 6.17 Message Maintenance Initial Screen

On the next screen, select the **Messages** tab to edit your messages. The system will prompt you to assign a package when navigating to the **Messages** tab because a message class is a repository object. Message texts are similar to text symbols; each message text is assigned a unique three-digit alphanumeric value.

You can select the **SelfExplanatory** checkbox (see Figure 6.18) if the message explains itself sufficiently, or you can create long texts for each message to show more information about a message to the user. The long text is displayed when the user

double-clicks the message (see Figure 6.19). The long text is maintained by clicking the **Long Text** button, as shown in Figure 6.18.

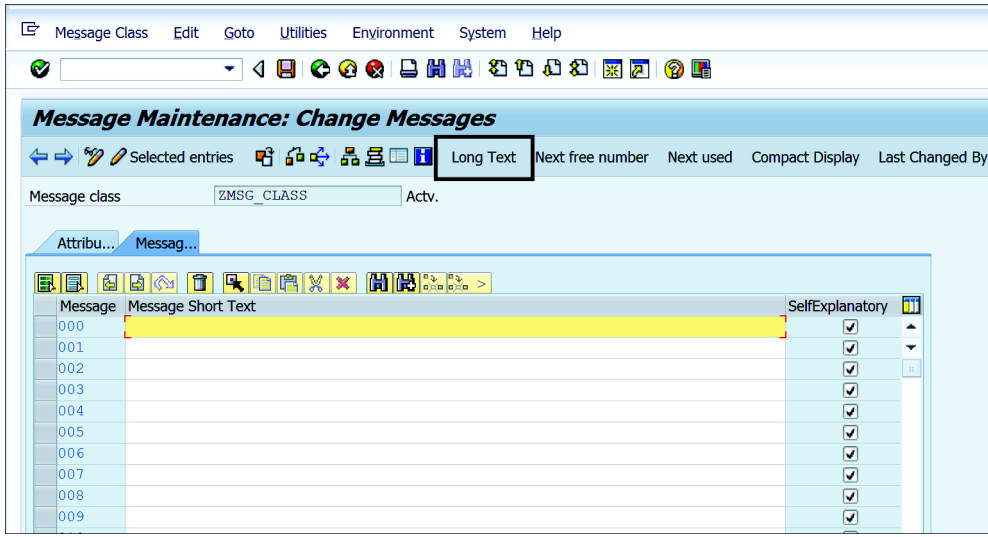


Figure 6.18 Maintaining Messages in a Message Class

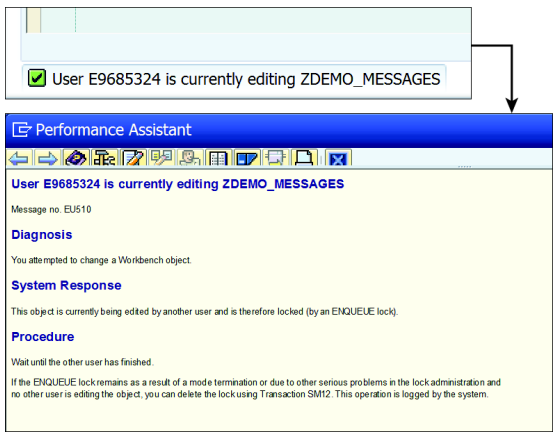


Figure 6.19 Long Text Displayed in the Performance Assistant Window

The syntax to display a message from the message class is a bit different from displaying it using a text symbol. Here, the message number is preceded by the type of

message as opposed to using the TYPE addition. The message class name is supplied in parenthesis, for example:

```
MESSAGE e000(ZMSG_CLASS).
```

In this statement, we're displaying the message with the message number 000, maintained in the message class ZMSG_CLASS as an error message. The message type e precedes the message number 000. You can use other message types similarly; for example, w000 will display the message as a warning message.

The message class name can also be maintained in the REPORT statement for executable programs, using the MESSAGE-ID addition, instead of maintaining it in parenthesis for each message, for example:

```
REPORT  ZDEMO_MESSAGES MESSAGE-ID zmsg_class.

MESSAGE E000.
```

You can maintain translations for each message by choosing **Goto • Translation** from the **Message Maintenance** screen.

6.2.4 Dynamic Messages

You may have noticed that Figure 6.19 contains some dynamic elements. The message shows the username and the program name as part of the message text. You can use an ampersand (&) as a placeholder in message text, and it can be replaced dynamically in the program via the WITH addition to the MESSAGE keyword. You can use up to four placeholders in the message text.

Figure 6.20 shows the message with two placeholders that will be replaced with a value supplied in the MESSAGE statement via the WITH addition.

The message can be sent using the syntax MESSAGE s000(ZMSG_CLASS) WITH sy-uname sy-repid. Here, sy-uname is a system field that stores the username, and sy-repid is the system field that stores the current program name. The value of the variable sy-uname will replace the first placeholder in the message, and sy-repid will replace the second placeholder. The fields are inserted sequentially into the message text in place of the placeholders.

Alternatively, you can use the placeholders with position numbers such as &1, &2, &3, and &4 in the message to specify where the value is inserted in the message. Placeholders that aren't supplied with a value via the WITH addition will be ignored.

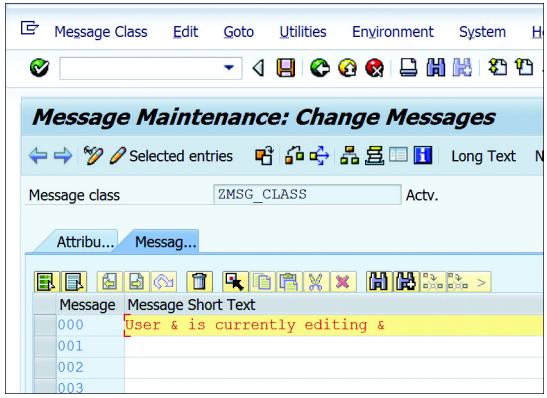


Figure 6.20 Using Placeholders in Message Text

Placeholders can also be used in the long text of a message by maintaining `&V1&` to `&V4&` as the placeholders. When the long text is called, the values supplied with the message will replace the placeholders in the long text.

6.2.5 Translation

Because the SAP system supports multiple logon languages, it's recommended to create translations for all texts maintained in the program (in case users with different logon languages access your program).

The translation editor can be accessed by choosing **Goto • Translation** from the ABAP Editor or from text maintenance screens like those for the message class or the text symbols. After selecting the **Goto • Translation**, the system presents a dialog box to specify the target language into which the text needs to be translated, as shown in Figure 6.21.

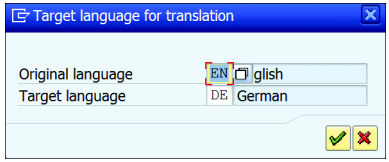


Figure 6.21 Target Language Selection

After the target language is selected, you can maintain translations manually for each text individually and click **Save**, as shown in Figure 6.22.

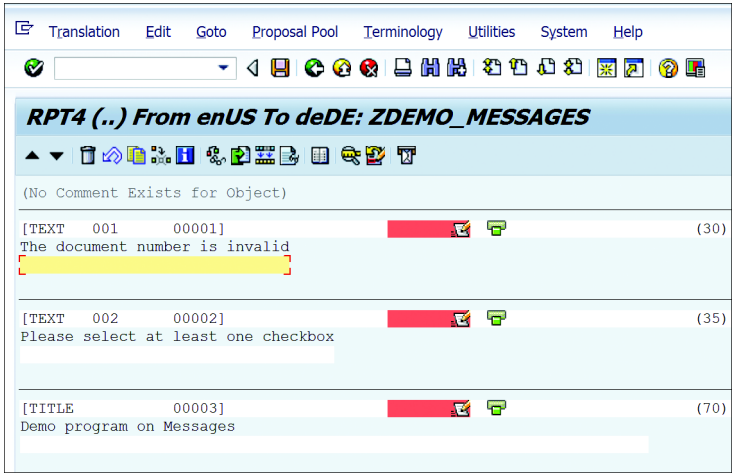


Figure 6.22 Maintain Translation Screen

Translations can be created for the program title, selection texts, text symbols, and list headings.

6.3 Summary

In this chapter, we discussed the concepts behind selection screens and messages. Understanding both these concepts is required to create your first complete report program. We also explored different ways to use the `PARAMETERS` and `SELECT-OPTIONS` keywords to create different selection screen elements. We've created the groundwork to create a report program, which we'll explore further with selection screens and their events in Chapter 14.

We also looked at messages and the different types of messages that can be sent from your ABAP programs. We touched on various ways to maintain messages and discussed both static and dynamic messages. Finally, we concluded by discussing translation. It's important to note that all texts can be translated to multiple languages.

We'll wrap up our discussion of ABAP basics in the next chapter, in which we'll discuss key concepts in modularization techniques. The concepts in the next chapter are crucial to understanding the advanced topics in ABAP.

Chapter 7
Modularization Techniques

ABAP programs are modular in nature. Taking advantage of modularization techniques will provide a number of advantages, including better readability and organization. In this chapter, we'll discuss ABAP modularization.

Modularization is breaking down and organizing program code into a modular structure, rather than dealing with the program as a single, indivisible unit. There are many advantages to modularization, including reusability of code, which helps in reducing code redundancy, and better organization of program code, which in turn makes your ABAP programs maintenance-friendly.

In this chapter, we'll discuss the ABAP program flow and runtime environment control program execution through events and procedures. We'll also look at the concept of processing blocks, including their features, the statements that define them, how they're called, whether they can keep local data, and ABAP programs in which they can be defined.

In Section 7.1, we'll begin with an overview of modularization in general. We'll introduce key concepts and provide an example to get you started. Section 7.2 then explores the program structure, including the use of event blocks, dialog modules, and procedures.

Section 7.3 dives into the different event type categories: program constructor events, reporting events, selection screen events, list events, and screen events. Then, in Section 7.4, we take a look at procedures—specifically, subroutines, function modules, and methods. Finally, we end the chapter in Section 7.5, using inline declarations for declaring data objects inline.

7.1 Modularization Overview

ABAP programs are modular in nature, allowing you to break down the program functionality into smaller modules. These modules are known as *processing blocks*. Processing blocks can be put together to achieve the overall functionality of a program.

To better illustrate this process, let's look at an example. Say you're developing an application for your HR department, and one of the requirements of the application is to find the day an employee joined the organization. This is just a *part* of the overall functionality the application provides, but this feature is required in many applications in the human resources (HR) functionality.

To find the weekday of a given date, you may have to write some 40 lines of code. If you're a new developer, you may spend a few hours or days getting the logic right. If you chose to implement the code directly in the main program, you'll have to repeat the code in all the other applications that require similar functionality.

Worst of all, if none of the other developers in your organization are aware that you've already spent hours or days implementing the logic for this requirement, they may try to implement the same logic on their own, which leads to multiple people spending time working on the same task.

By implementing this functionality in a module (let's say a *function module*) that can be called from any program, you avoid code redundancy and save a lot of time and effort.

Modularizing your ABAP programs leads to many benefits, including the following:

- Improves code readability
- Eases maintenance
- Helps develop reusable software blocks
- Allows the design of applications with scope for future enhancements
- Saves development effort
- Helps break down big, complex requirements into smaller, easy-to-maintain modules that can be combined to meet your requirements
- Splits development across team members without locking the code

As you spend more time developing ABAP programs, you may appreciate the advantages of modularization because it helps you simplify the development and maintenance of large applications.

In ABAP, we have various kinds of processing blocks. Some processing blocks are called by the ABAP runtime environment from outside the program, and some are called by ABAP statements from inside the program (called *procedures*).

Table 7.1 lists the various processing blocks and their key features.

Processing Block	Features
Called by the ABAP Runtime Environment	
Event blocks	Called by the ABAP runtime environment when a predefined event is triggered while the program is running. These processing blocks are maintained in the program using specific event keywords.
Dialog modules	Called by the ABAP runtime environment from the screen flow logic while processing general screens. These processing blocks are maintained in the main program of the screen between `MODULE...ENDMODULE` statements and are called from the screen flow logic using the `MODULE` statement.
Called by ABAP Statements	
Subroutines	Useful for modularization within the program. Although subroutines can be called both internally from ABAP statements within the program or externally from other programs, they are typically used for reusability within a program. Calling subroutines externally isn't recommended and should be avoided. Subroutines are maintained between `FORM...ENDFORM` statements and are called using the `PERFORM` statement.
Function modules	Procedures that are maintained globally in the system and can be called from any ABAP program. They are maintained using the Function Builder tool (Transaction SE37), which also serves as a central library to search for existing function modules. Function modules are maintained between `FUNCTION...ENDFUNCTION` statements in a *function pool* type of program and are called using the `CALL FUNCTION` statement. Although function modules work similarly to subroutines, they can only be defined globally, which in turn makes them easier to search for.

Table 7.1 Processing Blocks

Processing Block	Features
Methods	Contain the functions of classes in ABAP Objects. Methods are defined in classes and support object-oriented programming (OOP) concepts (see Chapter 8 for more information on OOP). Methods are maintained between METHOD...ENDMETHOD statements and are called using the CALL METHOD statement. Classes can be defined both locally within an ABAP program and globally in the Class Builder (Transaction SE24). ABAP Objects is an important topic that we'll discuss in detail in Chapter 8.

Table 7.1 Processing Blocks (Cont.)

In general, you use subroutines for reusability within a program, function modules for reusability across multiple programs in the system, and methods to implement OOP concepts along with reusability.

Modularization isn't just limited to processing blocks; you can also modularize your ABAP source code to organize the source code better with the help of *include programs* or *macros*:

- **Include programs**
  Include programs are repository objects for which the sole purpose is to modularize source code. Include programs are built into other programs and can't be executed independently.
- **Macros**
  Macros are another way to modularize code, but they're considered obsolete technology and shouldn't be used. Macros are difficult to debug but can be used to run a statement more than once in a code.

To better understand include programs, think of them this way: If you want to use a similar set of lengthy data declarations in multiple programs, you can keep the declarations in an include program that can be included in any ABAP program. This way, you can reuse the source code without having to type the same code in multiple programs. This also helps maintain the code centrally.

In ABAP Objects, include programs are useful to split up classes into visibility parts, declarations, methods, and so on. The sole purpose of the include program is to modularize the source code. The ABAP runtime environment doesn't recognize include programs as independent main programs; they're never directly executed and are used to be included in another main program.

Figure 7.1 shows the use of an include program.

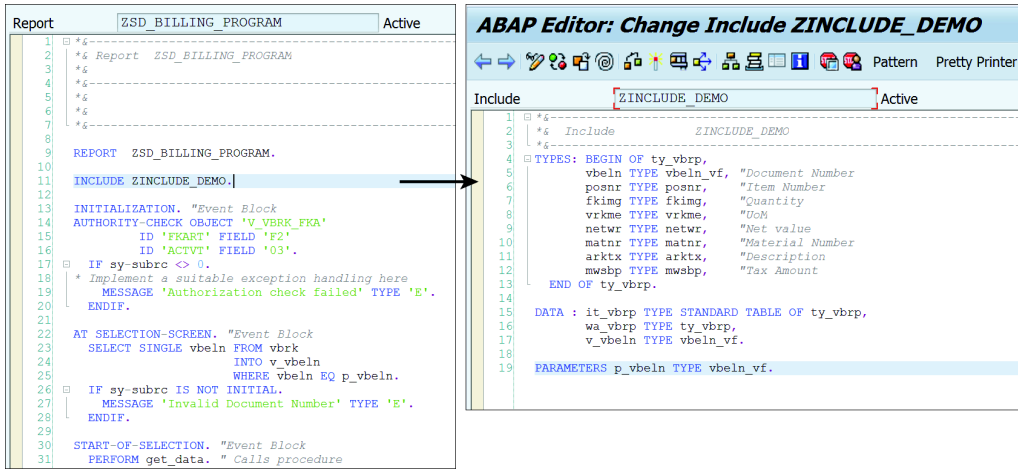


Figure 7.1 Include Program

Modularization in event blocks and dialog modules helps with the general execution of ABAP programs, and modularization in procedures and include programs helps improve the readability and maintainability of ABAP programs, allowing you to avoid code redundancies, to reuse functionality, and to encapsulate data.

Now that you have a basic understanding of the key concepts and methods used in modularization, we'll explore these techniques in greater depth in the remaining sections of this chapter.

7.2 Program Structure

ABAP was originally designed for the development of applications that process data from the database; users could interact with the application using SAP GUI screens. In a classic programming method based on screens (e.g., reports and transactions) that require user interaction, the control is passed back and forth between the presentation layer and the application layer. For example, when the user is making an input on the selection screen, the screen is active on the presentation layer, but there's no activity at this point in the application layer.

After the user clicks the **Execute** button, even though the selection screen is still displayed on the screen, it's not active. At this point, the application layer is actively processing the request, and the presentation layer is idle. After the application layer

processes the user request, the result is sent to the presentation layer. Now, the selection screen is active again, or a new screen is called.

The execution of the program between consecutive screens is called a *dialog step*. A dialog step in a report program starts when the user clicks the **Execute** button on the selection screen and ends when the results are displayed on the list screen. Again, after the user performs an action on the list screen, a new dialog step starts, which ends when a new screen is called and displayed to the user.

In such a scenario, you can't construct the program as a single sequential unit. You must divide the program into dialog steps, with each step comprising the programming logic between two successive screens.

Because ABAP programs are modular in structure, we divide the program into modules that can be assigned to the individual dialog steps. Each module is called a *processing block*, which consists of a set of ABAP statements. In other words, when you run a program, you're calling a series of processing blocks.

In this section, we'll discuss the use of processing blocks, event blocks, dialog modules, and procedures in the ABAP program structure.

7.2.1 Processing Blocks

In general, processing blocks can be broadly classified as event blocks, dialog modules, or procedures.

In the following subsections, we'll discuss the use of processing blocks in the program structure before discussing the different types of processing blocks that can be used.

Declaration Area

Each ABAP program consists of a global declaration area followed by a number of processing blocks. The objects defined in the global declaration part are visible in all processing blocks of the ABAP program. In some processing blocks (e.g., procedures), data declarations can be made that are locally visible.

Listing 7.1 shows an example of global and local declarations.

```
REPORT ZCB_EXAMPLE.
*Global declarations of the program
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
```

```
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
*Local declaration that can be accessed (visible) within
*this subroutine
  DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
  WRITE f2.
  WRITE f3.
ENDFORM.
```

Listing 7.1 Local and Global Data

In Listing 7.1, we defined data objects `f1` and `f2` in the global declaration area of the program. The program contains a subroutine with the name `OUTPUT`, within which we declared the data object `f3`. Because the data object `f3` is declared within the subroutine `OUTPUT`, it's local to this subroutine. In other words, the data object `f3` is locally visible within the subroutine `OUTPUT` and can be accessed only by ABAP statements that are part of the subroutine.

You can't access this data object `f3` from other processing blocks of the program. However, the global data objects `f1` and `f2` can be accessed by ABAP statements from any processing block of the program.

Using Processing Blocks

Processing blocks are indivisible units that contain ABAP statements. For example, in Listing 7.1, `END-OF-SELECTION` is an event block, which is a type of processing block that contains the ABAP statement `PERFORM`. `FORM OUTPUT...ENDFORM` is a subroutine, which is also a type of processing block that contains the `WRITE` statements.

> **Note**
>
> Processing blocks can't be nested, so you can't include one processing block in another processing block. For example, ahead in Listing 7.8, you can't include the `END-OF-SELECTION` processing block within the `FORM OUTPUT...ENDFORM` processing block. However, you can call one processing block from another processing block. In

Listing 7.1, we're calling the OUTPUT processing block from the END-OF-SELECTION processing block using the PERFORM statement.

Calling Processing Blocks

Processing blocks are called either externally by the ABAP runtime environment or internally by specific statements in the program. For example, END-OF-SELECTION is an event block called by the ABAP runtime environment when the program is running, and FORM OUTPUT is a procedure called by the PERFORM statement in the code.

Sequence of Processing Blocks

When we divide program code into various processing blocks, the program code isn't executed in the sequence in which the processing blocks are defined in the program. When a particular processing block is called depends totally on the ABAP runtime environment (e.g., for event blocks and modules) or when a specific processing block call statement is executed (e.g., for procedures).

Therefore, the order in which the processing blocks are maintained in the program source code is irrelevant to the sequence in which they're executed. After the processing block is called, the code in a processing block is executed sequentially.

For example, in Listing 7.1, the order in which the processing blocks are maintained in the code is irrelevant. Listing 7.2 shows the same code but rearranges the order of the END-OF-SELECTION and FORM OUTPUT processing blocks in the program; it returns the same output.

```
REPORT ZCB_EXAMPLE.
*Global declarations of the program.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
*Local declaration that can be accessed (visible) within
*this subroutine
  DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
```

```
  WRITE f2.
  WRITE f3.
ENDFORM.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
```

Listing 7.2 Changing the Processing Block Order

After executing the code in a processing block, control is returned to its caller.

Ending Processing Blocks

The execution of a processing block ends when the last statement of the processing block has been executed. However, it can also be ended programmatically by using certain statements such as CHECK, EXIT, or RETURN. For example, after checking for a certain condition, if you don't want to execute the remaining code in a processing block if the condition isn't met, then you can use the CHECK, EXIT, or RETURN statement to end the processing block without executing the remaining code in the processing block.

The CHECK and EXIT statements behave differently if they're used inside a loop statement (DO loop, WHILE loop, LOOP AT, etc.). When these statements are used within a loop, they don't end the processing block; they only affect the loop processing. For example, the CHECK statement will quit the current loop and process the next iteration when used in a loop.

However, the EXIT statement will quit the loop processing completely. Both the EXIT and CHECK statements will continue executing the remaining code in the processing block after loop processing. The RETURN statement always quits the processing block irrespective of whether it's used inside a loop or outside a loop. No matter how the processing block ends, control is returned to the caller of the processing block on each termination.

Listing 7.3 shows the usage of the EXIT statement within the subroutine OUTPUT. The statement WRITE f3 maintained after the EXIT statement won't be executed because the program will quit the processing block when it reaches the EXIT statement.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
```

```
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
  WRITE f2.
EXIT.
  WRITE f3.
ENDFORM.
```

Listing 7.3 Using the EXIT Statement within a Processing Block

Listing 7.4 shows an example using the `EXIT` statement inside a `DO` loop. After the first loop, the system will quit the loop when it encounters the `EXIT` statement but will continue executing the remaining code in the subroutine. Therefore, in this case, the statement `WRITE f3` will be executed.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
DATA f3 TYPE c VALUE 'C'.
DO 3 TIMES.
  WRITE f1.
  WRITE f2.
EXIT.
ENDDO.
  WRITE f3.
ENDFORM.
```

Listing 7.4 EXIT Statement within a Loop

Listing 7.5 shows an example using the `CHECK` statement to quit the processing block. In this example, we're using the `CHECK` statement to check that the contents of data objects `f2` and `f3` don't match.

If the condition is satisfied, the remaining code of the processing block will be executed; otherwise, the system will quit the processing block without executing further code in the processing block. In our example, the statement `WRITE f3` won't be executed because `f2` and `f3` have the same value.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
  WRITE f1.
  WRITE f2.
CHECK f2 <> f3.
  WRITE f3.
ENDFORM.
```

Listing 7.5 Using the CHECK Statement in a Processing Block

Listing 7.6 shows an example using the `CHECK` statement within a `DO` loop to check if the contents of data objects `f1` and `f2` are the same.

When the system encounters the `CHECK` statement, it will execute the statements after the `CHECK` statement only if the `CHECK` condition is satisfied. If the condition isn't satisfied, the system will skip to the next loop pass without executing further code in the current loop pass.

After the loop processing is complete, the system will continue executing the remaining code in the processing block. Therefore, in this example, the `WRITE f1` statement is executed three times, the `WRITE f2` statement is never executed, and the `WRITE f3` statement is executed once.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
```

```
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
DO 3 TIMES.
  WRITE f1.
CHECK f1 = f2.
  WRITE f2.
ENDDO.
  WRITE f3.
ENDFORM.
```

Listing 7.6 CHECK Statement within a Loop

Listing 7.7 shows an example using the `RETURN` statement within the `DO` loop. Because the `RETURN` statement will always quit the processing block irrespective of whether it's used within a loop or outside the loop, the `WRITE f1` statement is executed once, and the `WRITE f2` and `WRITE f3` statements are never executed.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&---------------------------------------------------------------------*
*&      Form  OUTPUT
*&---------------------------------------------------------------------*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
DO 3 TIMES.
  WRITE f1.
RETURN.
  WRITE f2.
ENDDO.
  WRITE f3.
ENDFORM.
```

Listing 7.7 RETURN Statement

Statements

All ABAP program statements, except global data declarations (also referred to as *virtual processing blocks*), belong to a processing block. If you write any statements between the global declaration area and a processing block (the statement isn't explicitly assigned to a processing block and isn't a declarative statement), they're automatically assigned to the beginning of the START-OF-SELECTION processing block. The START-OF-SELECTION processing block plays the role of a default processing block in executable programs.

Listing 7.8 contains sample code using various processing blocks. Here, we're showing item data for a given billing document number.

```
TYPES: BEGIN OF ty_vbrp,
       vbeln TYPE vbeln_vf, "Document Number
       posnr TYPE posnr,    "Item Number
       fkimg TYPE fkimg,    "Quantity
       vrkme TYPE vrkme,    "UoM
       netwr TYPE netwr,    "Net Value
       matnr TYPE matnr,    "Material Number
       arktx TYPE arktx,    "Description
       mwsbp TYPE mwsbp,    "Tax Amount
  END OF ty_vbrp.

DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
       wa_vbrp LIKE LINE OF it_vbrp,
       v_vbeln TYPE vbeln_vf.

PARAMETERS p_vbeln TYPE vbeln_vf.

INITIALIZATION. "Event Block
AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
         ID 'FKART' FIELD 'F2'
         ID 'ACTVT' FIELD '03'.
  IF sy-subrc <> 0.
* Implement a suitable exception handling here
    MESSAGE 'Authorization check failed' TYPE 'E'.
  ENDIF.

AT SELECTION-SCREEN. "Event Block
  SELECT SINGLE vbeln FROM vbrk
```

```
                      INTO v_vbeln
                      WHERE vbeln EQ p_vbeln.
  IF sy-subrc IS NOT INITIAL.
    MESSAGE 'Invalid Document Number' TYPE 'E'.
  ENDIF.

START-OF-SELECTION. "Event Block
  PERFORM get_data. "Calls procedure
  PERFORM show_output. "Calls procedure
*&---------------------------------------------------------------------*
*&      Form  GET_DATA
*&---------------------------------------------------------------------*
FORM get_data . "Subroutine
  SELECT vbeln
         posnr
         fkimg
         vrkme
         netwr
         matnr
         arktx
         mwsbp
               FROM vbrp
               INTO TABLE it_vbrp
               WHERE vbeln EQ p_vbeln.
ENDFORM.                    " GET_DATA

*&---------------------------------------------------------------------*
*&      Form  show-output
*&---------------------------------------------------------------------*
FORM show_output . "Subroutine
  FORMAT COLOR COL_HEADING.
  WRITE : / 'Item',
          10  'Description',
          33  'Billed Qty',
          48  'UoM',
          57  'Netvalue',
          70  'Material',
          80  'Taxamount'.
  FORMAT COLOR OFF.
```

```
  LOOP AT it_vbrp INTO wa_vbrp.
    WRITE :/  wa_vbrp-posnr,
          10  wa_vbrp-arktx,
          33  wa_vbrp-fkimg LEFT-JUSTIFIED,
          48  wa_vbrp-vrkme,
          57  wa_vbrp-netwr LEFT-JUSTIFIED,
          70  wa_vbrp-matnr,
          80  wa_vbrp-mwsbp LEFT-JUSTIFIED.
  ENDLOOP.
ENDFORM.                    " SHOW-OUTPUT
```

Listing 7.8 Code Showing Various Processing Blocks

In the report that results from Listing 7.8, the user inputs the document number of a billing document on a selection screen, and the item details are fetched from table VBRP based on that document number.

In this code, we're using five processing blocks: three event blocks (INITIALIZATION, AT SELECTION-SCREEN, START-OF-SELECTION) and two procedures (FORM get_data and FORM show_output).

The event blocks allow us to control the sequence in which our code is executed. For example, the code related to the authorization check is maintained under the INITIALIZATION event block (we can also use another event block, LOAD-OF-PROGRAM, as an alternative). Because this event is always triggered when the program is initialized, we can ensure that the first element checked when the user runs the report is whether that user has the required authorizations to run the report.

Similarly, the AT SELECTION-SCREEN event is triggered when the user clicks the **Execute** button on the selection screen. By keeping the code to validate the user input under this event block, we can ensure that the program execution doesn't continue without a valid document number.

We've also separated the selection logic from output logic by maintaining two procedures in the program: FORM get_data and FORM show_output. These subroutines are called by the PERFORM statements in the START-OF-SELECTION event block.

Thus far, we've looked at how processing blocks are used in an ABAP program structure. In doing so, we've already touched on the different types of processing blocks available. In the next section, we'll take a closer look at these processing block types.

Types of Processing Blocks

As previously mentioned, processing blocks can be classified as event blocks, dialog modules, or procedures. The differentiating factors among the types of processing blocks lie in how they're introduced in the program and how they're called. The type of processing block also establishes whether it can keep local data and its support for the parameter interface.

Notice that in Listing 7.8, although procedures are closed with their own `END` statements (`FORM...ENDFORM`), event blocks are introduced with event keywords and end implicitly by the beginning of the next processing block, by the end of the program, or with the first definition of a modularization unit. In addition, the event blocks are called by the ABAP runtime environment automatically when the program is running. Alternatively, subroutines (procedures) are called with the specific `PERFORM` keyword in the code under the `START-OF-SELECTION` event block.

> **Note**
>
> As discussed earlier, you can call processing blocks from outside the ABAP program or by using ABAP commands (which are themselves part of a processing block).

Event blocks behave differently from other processing blocks. Event blocks are always called by the ABAP runtime environment, and the call is triggered by an event. For example, user actions on selection screens and lists trigger certain events for which corresponding event blocks are called. The ABAP runtime environment also triggers events (while the ABAP program is running) that can be processed in ABAP programs.

For example, an event triggers when the program is loaded into the memory, before a selection screen is sent. You can make the program react to these events by defining the corresponding event blocks in your program and ignoring an event block if you don't want the program to react to that event. This is unlike a subroutine call, where you must have a corresponding subroutine. In other words, you aren't forced to react to a particular event because the event is simply ignored if a corresponding event block isn't maintained in your ABAP program while allowing you to react to an event by maintaining the event block.

Note that event blocks and dialog modules don't have a local declaration area. Any data declarations within these blocks are automatically added to the global declaration area of the program, making these blocks behave like global data (visible in all

internal processing blocks). However, any data declarations within procedures (method, subroutines, function modules) are called *local data declarations* and are only visible within the procedure in which they are declared. They are created when the procedure starts and destroyed when the procedure ends.

Processing blocks can be maintained in any order in the program. The code in Listing 7.9 shows the same code as in Listing 7.8, but the order of the processing blocks is changed to highlight the fact that the order of the processing blocks in the program is irrelevant.

```
TYPES: BEGIN OF ty_vbrp,
       vbeln TYPE vbeln_vf, "Document Number
       posnr TYPE posnr,    "Item Number
       fkimg TYPE fkimg,    "Quantity
       vrkme TYPE vrkme,    "UoM
       netwr TYPE netwr,    "Net Value
       matnr TYPE matnr,    "Material Number
       arktx TYPE arktx,    "Description
       mwsbp TYPE mwsbp,    "Tax Amount
  END OF ty_vbrp.

DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
       wa_vbrp TYPE ty_vbrp,
       v_vbeln TYPE vbeln_vf.
PARAMETERS p_vbeln TYPE vbeln_vf.

AT SELECTION-SCREEN. "Event Block
  SELECT SINGLE vbeln FROM vbrk
                      INTO v_vbeln
                      WHERE  vbeln EQ p_vbeln.
  IF sy-subrc IS NOT INITIAL.
    MESSAGE 'Invalid Document Number' TYPE 'E'.
  ENDIF.

START-OF-SELECTION.
  PERFORM get_data. "Calls procedure
  PERFORM show_output. "Calls procedure

INITIALIZATION. "Event Block
AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
```

```
        ID 'FKART' FIELD 'F2'
        ID 'ACTVT' FIELD '03'.
  IF sy-subrc <> 0.
* Implement suitable exception handling here
    MESSAGE 'Authorization check failed' TYPE 'E'.
  ENDIF.

*&---------------------------------------------------------------------*
*&      Form  show-output
*&---------------------------------------------------------------------*
FORM show_output . "Subroutine
  FORMAT COLOR COL_HEADING.
  WRITE : / 'Item',
         10  'Description',
         33  'Billed Qty',
         48  'UoM',
         57  'Netvalue',
         70  'Material',
         80  'Taxamount'.
  FORMAT COLOR OFF.
  LOOP AT it_vbrp INTO wa_vbrp.
    WRITE :/  wa_vbrp-posnr,
           10  wa_vbrp-arktx,
           33  wa_vbrp-fkimg LEFT-JUSTIFIED,
           48  wa_vbrp-vrkme,
           57  wa_vbrp-netwr LEFT-JUSTIFIED,
           70  wa_vbrp-matnr,
           80  wa_vbrp-mwsbp LEFT-JUSTIFIED.
  ENDLOOP.
ENDFORM.                    " SHOW-OUTPUT

*&---------------------------------------------------------------------*
*&      Form  GET_DATA
*&---------------------------------------------------------------------*
FORM get_data . "Subroutine
  SELECT vbeln
         posnr
         fkimg
         vrkme
```

```
          netwr
          matnr
          arktx
          mwsbp
                FROM vbrp
                INTO TABLE it_vbrp
                WHERE vbeln EQ p_vbeln.
ENDFORM.                    " GET_DATA
```

Listing 7.9 Code from the Previous Listing, Rearranged to Change Order of Processing Blocks

The code in Listing 7.8 and Listing 7.9 behave the same and produce the same results (see Figure 7.2).

Demo Program

Item	Description	Billed Qty	UoM	Netvalue	Material	Taxamount
000010	Sony Sunny01	5,000	PC	8.150,00	M-01	1.222,50
000020	Sony Xa1	5,000	PC	9.440,00	M-02	1.416,00
000030	Highscreen MS 1775P	4,000	PC	9.400,00	M-10	1.410,00
000040	MAG DX 15F/Fe	4,000	PC	5.848,00	M-12	877,20

Figure 7.2 Output of Code in Listing 7.8 and Listing 7.9

The following sections discuss the different processing blocks in greater depth.

7.2.2 Event Blocks

Event blocks are introduced in the program through event keywords. Each event has a specific keyword, and you can use the respective event keyword in the program to make it react to that event. When the event is triggered by the ABAP runtime environment, the code under the event keyword is executed.

An event block ends implicitly when the next processing block starts, as opposed to specifically by its own keyword. Therefore, unlike procedures, we don't have to use any `END` statement to end the event block explicitly.

Event blocks are internal to the program and can't be called programmatically using any ABAP statements. They're always triggered by events in the ABAP runtime environment. They don't support local data or parameter interfaces. When you use

declarative statements within an event block, the system won't throw any syntax errors, but rather treats the declarative statement as a global declaration.

If you maintain an appropriate event block in the ABAP program for an event of the ABAP runtime environment, the code in the event block is executed; otherwise, the event is simply ignored.

Conversely, the code in an event block won't be executed unless the associated event takes place while the program is being executed. In other words, just because you've maintained an event block in your program, that doesn't guarantee its execution. It's only executed if the associated event is triggered while the program is running.

For example, the `AT LINE-SELECTION` event is triggered when the user double-clicks any output line on the list screen. If you want your program to react to the double-click event, you can maintain this event keyword in your program and write the corresponding code under it.

When the user double-clicks the output line, the code under the `AT LINE-SELECTION` event block is executed. However, if the user doesn't double-click the output line, this code is never executed. For example, in the following code, the statement `CALL TRANSACTION 'MM03'` will be executed when the user double-clicks a line in the list output; if the user doesn't double-click, this statement is never executed:

```
AT LINE-SELECTION.
CALL TRANSACTION 'MM03'.
```

We'll discuss all the available events and their keywords in Section 7.3.

7.2.3 Dialog Modules

Dialog modules are implemented in executable programs, module pool programs, or function groups between `MODULE` and `ENDMODULE`.

You use dialog modules when working with dialogs (screens). Because executable programs, module pools, and function groups are the only programs that support screens, you can use dialog modules only in these type of programs. Note that you can't define screens and modules with ABAP Objects.

You maintain dialog modules in the main program where a screen is defined and call these modules from the screen flow logic using the `MODULE` keyword, as shown in Figure 7.3.

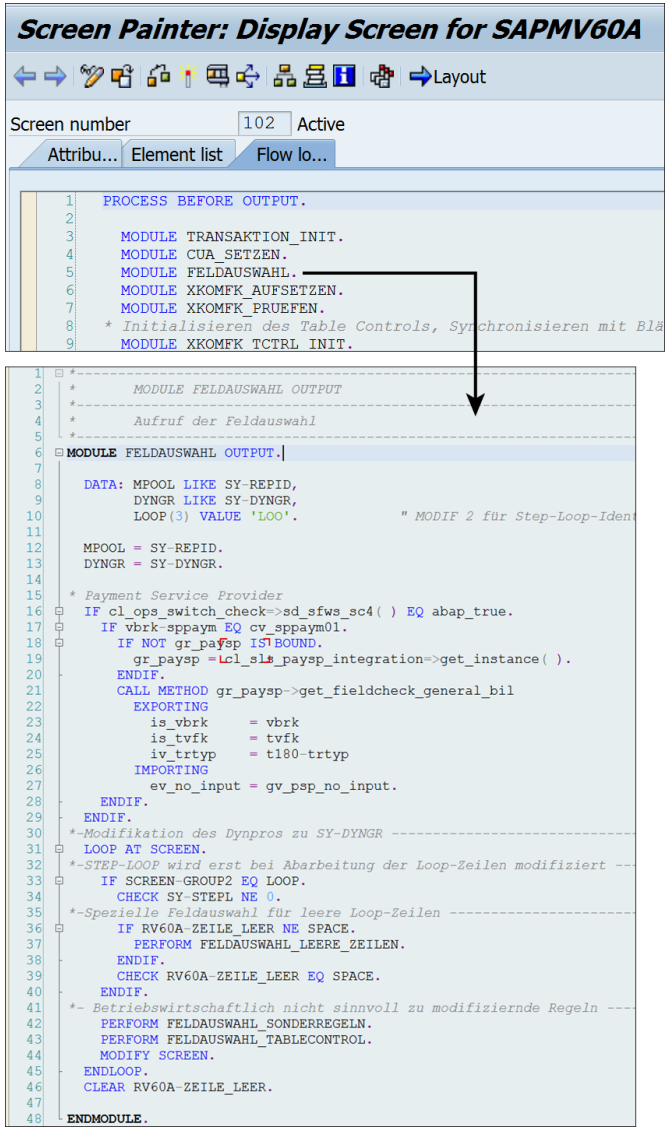


Figure 7.3 Module Call

Like event blocks, dialog modules have no local data area and no parameter interface. A data declaration in a dialog module is also added to the global data.

We'll discuss dialog modules further when we cover dialog programming in Chapter 12.

7.2.4 Procedures

Procedures are callable processing blocks; that is, they're called by ABAP statements in the program. Procedures support a mechanism—the parameter interface—to import and export data for the calling program and to contain a local data area. You can exchange data with the procedure using its parameter interface. They can be called internally and externally, meaning that they can be called from within the same program or externally from other ABAP programs.

The main purpose of procedures is to create reusable software blocks. For example, say you need a specific function in many programs or, many times, within the same program. You can simply create a procedure and call it where you need that functionality, instead of manually developing the same program logic many times. SAP provides many standard procedures out of the box that can be called in your ABAP programs.

Understanding procedures is crucial because doing so helps you take advantage of many SAP-delivered reusable libraries and lays the foundation to grasp certain advanced concepts in ABAP, such as Business Application Programming Interfaces (BAPIs), forms, SAP List Viewer (ALV), and so on. For example, BAPIs are technically implemented as remote function calls (RFCs), so developing a BAPI requires an understanding of function modules. Similarly, an understanding of ABAP Objects helps you work with ALV object models.

7.3 Events

Events can be broadly classified into the following categories:

- Program constructor events
- Reporting events
- Selection screen events
- List events
- Screen events

The next sections look at each of these event types.

7.3.1 Program Constructor Events

The `LOAD-OF-PROGRAM` event block serves as a *program constructor*. A program constructor serves the same purpose for ABAP programs as a constructor serves for a class (i.e., to initialize program variables).

This type of event block is called by the ABAP runtime environment when an executable program, module pool program, function group, or subroutine pool is loaded into memory (internal session). This event is triggered only once per program execution. If the program call opens a new internal session for each call (e.g., when using the SUBMIT or CALL TRANSACTION statements to call another program), then the LOAD-OF-PROGRAM event is triggered once per every program call.

To make the program react to this event, we use the LOAD-OF-PROGRAM event keyword, which is the first event to be called before any other ABAP code is processed. The statements in this block allow you to initialize the data of the program.

The LOAD-OF-PROGRAM event is mainly used when calling external procedures to initialize the global data of the called procedure. For example, in Listing 7.10 the data object DATE is initialized with the current date via the LOAD-OF-PROGRAM event.

By maintaining the statement date = sy-datum under the LOAD-OF-PROGRAM event, we ensure that the data object DATE is always initialized with the current date when the program is executed.

```
REPORT ZCB_EXAMPLE.
DATA date TYPE d.
LOAD-OF-PROGRAM.
date = sy-datum.
```

Listing 7.10 Using LOAD–OF–PROGRAM to Initialize Data

7.3.2 Reporting Events

Reporting events can be used in executable programs (report programs). Executable programs are event-driven. When you run an executable program, a series of events are triggered, one after the other, by the ABAP runtime environment. You can define event blocks to react to these events. Reporting events that can be used in an executable type of program include the following:

- INITIALIZATION
- START-OF-SELECTION
- GET table
- GET table LATE
- END-OF-SELECTION

In the following subsections, we'll take an in-depth look at each of these reporting events.

INITIALIZATION

INITIALIZATION is the first reporting event triggered when an executable program is running. If the program also has a LOAD-OF-PROGRAM event block, then the INITIALIZATION event block is executed immediately after the LOAD-OF-PROGRAM event block.

If the executable program has a selection screen defined, an INITIALIZATION event is called after the ABAP selection screen code has been processed but before the selection screen is displayed to the user.

Therefore, you can use it to initialize input fields of the selection screen or change the default values of these at runtime before the user gets to enter data into them, as shown in Listing 7.11.

```
PARAMETERS p_carrid TYPE s_carr_id DEFAULT 'LH'.
INITIALIZATION.
p_carrid = 'AA'.
```

Listing 7.11 INITIALIZATION Event

However, an INITIALIZATION event is triggered only once per program execution, so any subsequent call to the selection screen won't execute the code under the INITIALIZATION event. It just works when the selection screen is called by the ABAP runtime environment for the first time. For this reason, we recommend using selection screen-related events to initialize or modify the selection screen fields and using the INITIALIZATION event to initialize all the values in the program.

In Listing 7.11, the p_carrid parameters field is assigned a default value LH. However, this value can be changed at runtime via the INITIALIZATION event; as a result, the user will see AA as the default value on the selection screen.

START-OF-SELECTION

The START-OF-SELECTION event is called after the INITIALIZATION event, when an executable program is running. If the program has a selection screen defined, this is the first event that is triggered after all selection screen processing has been completed (after executing all selection screen events). Selection screen events are executed between INITIALIZATION and START-OF-SELECTION events.

The START-OF-SELECTION event plays the role of a standard event, which means that if no other events are declared, there's no need to code this event manually. All processing statements will be automatically assigned to an implicit START-OF-SELECTION block, as shown in Figure 7.4.

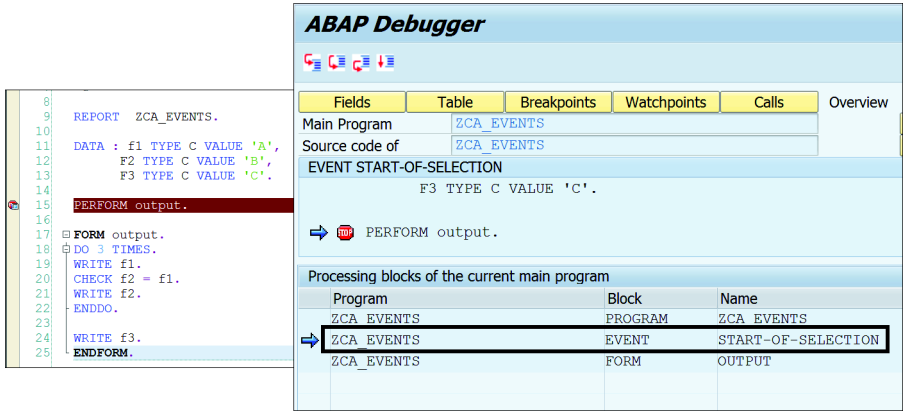


Figure 7.4 START-OF-SELECTION as Standard Event

Similarly, if you have any statements between the global declaration area and the first processing block of the program, they're automatically inserted at the beginning of this event block and executed.

In Figure 7.5, the statement at line **28** isn't explicitly assigned to any event, so the system automatically adds it to the top of the `START-OF-SELECTION` event and executes it before the `PERFORM` statement at line **46**.

```
22   DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
23          wa_vbrp TYPE ty_vbrp,
24          v_vbeln TYPE vbeln_vf.
25
26   PARAMETERS p_vbeln TYPE vbeln_vf.
27
28   CLEAR wa_vbrp.
29
30   INITIALIZATION. "Event Block
31   AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
32            ID 'FKART' FIELD 'F2'
33            ID 'ACTVT' FIELD '03'.
34     IF sy-subrc <> 0.
35   * Implement a suitable exception handling here
36       MESSAGE 'You do not have the authorization to run the report' TYPE 'E'.
37     ENDIF.
38
39   AT SELECTION-SCREEN. "Event Block
40     SELECT SINGLE vbeln FROM vbrk INTO v_vbeln WHERE  vbeln EQ p_vbeln.
41     IF sy-subrc IS NOT INITIAL.
42       MESSAGE 'Invalid Document Number' TYPE 'E'.
43     ENDIF.
44
45   START-OF-SELECTION.
46     PERFORM get_data. " Calls procedure
47     PERFORM show_output. "Calls procedure
48
49
```

Figure 7.5 START-OF-SELECTION as Default Event

In general, you put all the data selection logic of your ABAP program within the START-OF-SELECTION event. Here, the program starts selecting values from tables. In a typical report program, you initialize the program data under the INITIALIZATION event, validate the selection screen inputs under the selection screen events, and, if everything looks fine, proceed with the data selection (based on the selection criteria) under the START-OF-SELECTION event.

GET Table

The GET table event is used when the executable program uses a logical database for data selection. A *logical database* is a special ABAP program that retrieves data from the database and makes it available to the application program. In other words, when you use a logical database in your program, you don't need to code the SQL statements to retrieve the data from the database.

The logical database itself contains the Open SQL statements that retrieve the data from the database and pass the data line by line to the *interface work area* of your application program.

A logical database consists of a program with selection screen and authorization checks. When a logical database is used for the report program, the program can automatically inherit the selection screen and authorization checks.

Interface work areas are defined using the TABLES or NODES keywords, and they work similarly to the work areas defined using the DATA keyword, with just one basic difference. These interface work areas are created in the shared data area between the programs. In other words, when you define a work area using the DATA keyword, it's defined in the memory space of the program. Interface work areas are defined in the shared memory space of the session, which all the programs running in the same session can access. For this reason, using TABLES isn't recommended, as it can produce side effects.

In older versions of the SAP ERP system, the only way to use a logical database was to link it to an executable program and use GET events to retrieve the data. However, since release SAP R/3 4.5A, it's been possible to call a logical database in any program using the LDB_PROCESS function module. This module allows you to call many logical databases in your program, nested in any order. In this respect, the use of interface work areas is practically obsolete, available only for backward compatibility with earlier releases.

The GET event is triggered when the logical database has read a line from the table mentioned in the event keyword and made it available to the program in the work area declared using the TABLES <table> or NODES <table> statement (as seen in Listing 7.12).

```
NODES: SPFLI.
GET SPFLI.
 SKIP.
 WRITE: / 'From:', SPFLI-CITYFROM,
 'TO :', SPFLI-CITYTO.
```

Listing 7.12 Code Showing the GET Event

GET <table> LATE

After all the data records for a node of the logical database have been read, the ABAP runtime environment triggers the GET <table> LATE event. By maintaining this event in the program, you can process the steps that should be executed at the end of the block. For example, if you want to print an itemized summary of sales orders, you can do so in this event block.

Listing 7.13 shows the implementation of this event block.

```
NODES: SPFLI, SFLIGHT, SBOOK.
DATA WEIGHT TYPE I VALUE 0.
GET SPFLI.
SKIP.
WRITE: / 'Carrid:', SPFLI-CARRID,
'Connid:', SPFLI-CONNID,
/ 'From: ', SPFLI-CITYFROM,
'To: ', SPFLI-CITYTO.
ULINE.
GET SFLIGHT.
SKIP.
WRITE: / 'Date:', SFLIGHT-FLDATE.
GET SBOOK.
WEIGHT = WEIGHT + SBOOK-LUGGWEIGHT.
GET SFLIGHT LATE.
WRITE: / 'Total luggage weight =', WEIGHT.
ULINE.
CLEAR WEIGHT.
```

Listing 7.13 Code Showing GET <table> LATE Event

In Listing 7.13, we're printing the date-wise summary of the total luggage weight for each airline code (CARRID) and flight connection number (CONNID). We retrieve the flight details using the GET SPFLI event. The GET SFLIGHT event loops through all the flights for a given date (the user can restrict the range on the logical database selection screen).

The GET SBOOK event retrieves the luggage weight for each booking from table SBOOK for the respective flight. Within this event, we're aggregating the total weight for the given date.

Finally, the GET SFLIGHT LATE event is called at the end after all the records for the flights in a given date are processed. In this event, we're printing the total luggage weight for the given date for the respective CARRID and CONNID. For this example, we're using the logical database selection F1S, which you can check using Transaction SLDB.

END-OF-SELECTION

The END-OF-SELECTION event is used to mark the end of selection. If your report isn't using a logical database, then this event is used after a START-OF-SELECTION event to mark the end of selection and to continue with further processing of the selected data, such as printing the selected data in the output. However, if your report is linked to a logical database, this event is called after the logical database has completely transferred all the records to the application program.

For example, in Listing 7.13, if all the data from the logical database selection is updated in an internal table that you then want to process further, you can use the END-OF-SELECTION event to mark the end of data selection and implement the logic to further process the selected data. The END-OF-SELECTION event is called by the runtime environment after processing all the GET events.

Using this event becomes mandatory to separate the selection logic from the output logic when using a logical database. Otherwise, its purpose is purely for documentation and for making the code legible. We recommend using this event to mark the separation from selection logic to output logic even if no logical database is used in the program.

Listing 7.14 shows an example using the END-OF-SELECTION event.

```
NODES: SPFLI, SFLIGHT, SBOOK.
DATA: gr_table TYPE REF TO cl_salv_hierseq_table.
DATA: it_spfli TYPE TABLE OF spfli.
DATA: it_sflight TYPE TABLE OF sflight.
```

```
DATA: it_binding TYPE salv_t_hierseq_binding.
DATA: wa_binding  LIKE LINE OF it_binding.

GET SPFLI.
APPEND spfli to it_spfli.
GET SFLIGHT.
APPEND sflight to it_sflight.

END-OF-SELECTION.
wa_binding-master = 'CARRID'.
wa_binding-slave = 'CARRID'.
APPEND wa_binding TO it_binding.

wa_binding-master = 'CONNID'.
wa_binding-slave = 'CONNID'.
APPEND wa_binding TO it_binding.

cl_salv_hierseq_table=>factory(
EXPORTING
  t_binding_level1_level2 = it_binding
IMPORTING
  r_hierseq = gr_table
CHANGING
  t_table_level1 = it_spfli
  t_table_level2 = it_sflight ).
gr_table->display( ).
```

Listing 7.14 END-OF-SELECTION Event

In Listing 7.14, we're filling the internal tables `IT_SPFLI` and `IT_SFLIGHT` within the `GET SPFLI` and `GET SFLIGHT` events, respectively. We're then displaying this data as a hierarchical sequential display using the ALV object model within the `END-OF-SELECTION` event. The ALV object model makes it easy to generate the ALV output. We'll discuss the ALV object model further in Chapter 15.

7.3.3 Selection Screen Events

Selections screens are special screens that allow the user to input the selection criteria for a program. These are typically used in report programs (executable programs), but they can also be used in module pool programs or function groups.

Selection screen events typically help to prepare the selection screen and evaluate user actions on the screen. For example, you can use selection screen events to assign default values to the input fields at runtime, dynamically disable or enable certain fields, or validate user input and show appropriate messages to the user.

Selection screen events are triggered when a selection screen is defined for the program and is processed by the ABAP runtime environment. If there is no selection screen defined for the program, none of the selection screen events are triggered.

Selection screens are categorized as special screens that can be defined using ABAP statements (`PARAMETERS` or `SELECT-OPTIONS`) and are generated by the ABAP compiler automatically, rather than defined manually using the Screen Painter (e.g., general screens).

Because selection screens are generated automatically, programmers don't have access to the screen flow logic in which they can define dialog modules to modify the screen or react to user actions. The ABAP runtime environment fully controls the screen processing, generating a number of selection screen events through which programmers can maintain the event blocks in the program to react to these events.

Screen flow logic is broadly divided into *process before output* (PBO) and *process after input* (PAI). PBO is triggered before the screen is displayed to the user, and PAI is triggered after user takes some action on the displayed screen (e.g., clicking a button). The ABAP runtime environment triggers different events at PBO and PAI while processing the screen.

For example, `AT SELECTION-SCREEN OUTPUT` is a PBO event; you can edit this event block in the program to react to the event and execute the code to modify the screen or to assign default values. Similarly, `AT SELECTION-SCREEN` is a PAI event that is triggered after the user takes some action on the screen. You can use this event block to validate user inputs.

The basic form of the selection screen events is the `AT SELECTION-SCREEN` event. This event is triggered after the ABAP runtime environment has transferred all input data from the selection screen to the ABAP program.

We'll explore all the available selection screen events in Chapter 14.

7.3.4 List Events

List screens are one of the three types of screens in the SAP system (along with dialog screens and selection screens). They are used to display output data and to support

user interaction. Like selection screens, list screens are also created using ABAP statements. In addition, the output of the list screen can be sent to a printer.

List screens are generated by using the `WRITE`, `ULINE`, or `SKIP` statements, and the ABAP runtime environment calls list events while processing the list screen. List screens provide a freely definable area that can be filled with data. List screens are generally used to display report output; however, this is considered old-fashioned, and these screens mostly have been replaced by ALV technology in newer systems.

Because the only three types of programs that support screens are executable programs, module pool programs, and function groups, list events can be used only in these types of programs while a classical list is being generated or when the user performs an action on the displayed list.

There are five list events:

- `TOP-OF-PAGE`
- `TOP-OF-PAGE DURING LINE-SELECTION`
- `END-OF-PAGE`
- `AT USER-COMMAND`
- `AT LINE-SELECTION`

`TOP-OF-PAGE` and `END-OF-PAGE` events are triggered in PBO and can be used to print the header (top) and footer (end) information on the page.

`AT USER-COMMAND` and `AT LINE-SELECTION` are triggered in PAI and can be used to react to user actions on screen. `TOP-OF-PAGE DURING LINE-SELECTION` is triggered when preparing the secondary list. For example, `AT USER-COMMAND` is triggered when the user selects any of the application toolbar buttons, whereas `AT LINE-SELECTION` is triggered when the user double-clicks the output line. These events can be used to provide drilldown reports.

We'll discuss all the available list events in detail in Chapter 13 when we cover list screens.

7.3.5 Screen Events

Screen events are triggered when processing general screens. *Special screens* (selection screens and list screens) are defined in ABAP programs using ABAP statements, but *general screens* are created using the Screen Painter. General screens are the most common screens used in transactions.

A general screen consists of the input/output fields and the screen flow logic. Like selection screens and list screens, general screens can be defined for executable programs, module pool programs, and function groups. General screens are commonly used with module pool programs.

The screen flow logic is divided into the PBO event, which is processed before the screen is displayed, and the PAI event, which is processed after a user action on the screen.

There are four screen events triggered when the screen is processed:

- **PBO**
  A PBO event is triggered before the screen is shown to the user and can be used to modify the screen.
- **PAI**
  A PAI event is triggered when the user performs an action on the screen. This event can be used to evaluate user actions.
- **Process on help request (POH)**
  A POH event is triggered when the user selects F1 help on the screen field. This event can be used to show field documentation to help the user learn more about a field.
- **Process on value request (POV)**
  POV is triggered when the user selects F4 help on the screen field. This event can be used to show the possible entries for a field (a value list).

All the screen events other than PBO are triggered after showing the program screen.

We'll discuss all the available screen events in more detail in Chapter 12 when we discuss dialog programming.

7.4 Procedures

Procedures are special modularization units that are called with ABAP statements and provide reusable software blocks. Procedures contain a set of statements that are executed when called from ABAP programs.

As shown in Figure 7.6, the ABAP program calls a procedure using specific ABAP statements (depending on the type of procedure being called). When the procedure is called, the code in the procedure is executed.

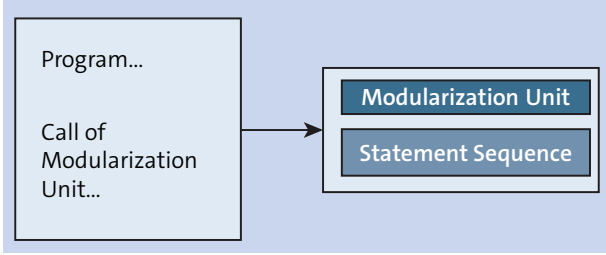


Figure 7.6 Call of Modularization Unit

Procedures are defined in ABAP programs, and upon activation of the program, they remain as standalone modules. As discussed previously, procedures can be called from the same program in which they're maintained or from external programs. For example, you can call a subroutine of the same program using the `PERFORM` statement, as shown:

```
PERFORM prepare_data.
```

If the subroutine is maintained in a different program and you want to call it externally, you can do so by giving the program reference. The following statement calls an external subroutine (note that it isn't recommended to call subroutines externally because they're not designed for external calls; for external calls, use function modules or ABAP classes):

```
PERFORM prepare_data IN PROGRAM ZCA_EXAMPLE.
```

When the procedure is called externally, the entire program in which the procedure is maintained is loaded into memory, but only the code inside the procedure is executed. Because procedures support the parameter interface (an interface to exchange data with the calling program), you can exchange data with the procedure to pass an input and receive a result.

Procedures support the local data area, the life of which is limited to the execution of the procedure. The data objects declared in the local data area of a procedure are created when the procedure is called and terminated after the execution of the procedure is completed.

For example, in the following code, the data object `f1` is defined within the subroutine `OUTPUT`. This data object `f1` is created when the subroutine is called and can be accessed by ABAP statements within the subroutine. After the subroutine execution

is completed (i.e., ENDFORM is executed), the data object f1 is no longer available in the program memory and can't be accessed by any ABAP statements.

```
FORM output.
DATA f1 TYPE C.
ENDFORM.
```

Any data stored in the local data is cleared upon termination of the procedure. Local data objects are useful when you don't want to occupy the memory space of the main program unnecessarily, especially after the purpose of the data has been fulfilled.

For example, say you're writing code to select data from two different database tables, copying it into two internal tables, and finally consolidating the data to a final internal table. After the data from these two internal tables is consolidated into the final table, you no longer need the data in the original internal tables.

If you defined these tables in the global declaration area, they continue to occupy memory throughout the runtime of the program, unless you manually clear the tables. However, if those internal tables are defined locally in a procedure, they're automatically cleared after the program exits the procedure.

ABAP contains the following kinds of procedures:

- **Subroutines**
  Subroutines are primarily used for local modularization. Even though they can be called externally from other ABAP programs, they're generally called locally from the program from which they're defined. Subroutines are useful for reusability within the program, and they can be defined in any ABAP program.
- **Function modules**
  Function modules are created globally and are used for global modularization. They're typically employed for reusability across multiple programs and are always called from a different program. Function modules are defined in a function group type of program but can be called from any ABAP program. They also support the RFC interface system and thus can be executed from remote systems.
- **Methods**
  Methods support OOP concepts and contain the functions of classes and their instances in ABAP Objects. Methods are defined in classes and can be maintained locally in the program or globally in the system.

In the following subsections, we'll take a closer look at each of these different ABAP procedure types.

7.4.1 Subroutines

Subroutines are maintained between FORM and ENDFORM statements and are called using the PERFORM statement:

```
FORM subr.
…
ENDFORM.
```

The additions USING and CHANGING define the parameter interface for the subroutine. The parameter interface allows you to pass data to a subroutine and is defined by directly declaring variables on the FORM statement; these variables are called *formal* parameters. Formal parameters are filled by the PERFORM statement by maintaining the parameters in the PERFORM statement; such parameters are called *actual parameters*.

Parameters can be defined by referring either to local data types or to global types in the ABAP Data Dictionary. The parameters defined exist as local variables to the subroutine; they're created when the subroutine is called and freed when it ends. If the subroutine has variables defined on the FORM statement, the PERFORM statement must pass a value to each of these variables in the defined sequence.

The parameter names that appear at the FORM level are called formal parameters, and the parameters that appear at the PERFORM level are called actual parameters.

The number of actual and formal parameters should always be the same. For example, if a subroutine has three formal parameters, then it expects three actual parameters to be passed when the subroutine is called using the PERFORM statement. In Listing 7.15, actual parameters f1, f2, and f3 are passed to formal parameters p1, p2, and p3.

```
PERFORM abc USING f1 f2 f3.

FORM abc USING p1 p2 p3.

ENDFORM.
```

Listing 7.15 Example Syntax for the Parameter Interface

While defining formal parameters, they can be either typed or untyped. If a formal parameter is referred to a data type, it's called a *typed formal parameter*; otherwise, it's an untyped formal parameter. In Listing 7.14, all parameters are untyped.

When an *untyped formal parameter* is used, you can pass a variable of any data type or length to it. In such a case, the formal parameter uses the attributes of the actual

parameter. For example, if you pass a 4-byte integer to an untyped formal parameter `p1`, then `p1` becomes a 4-byte integer. If you pass a character string of length three to the same parameter, it will become a three-character field. The data type and length of the untyped parameter is determined in runtime based on the actual parameter passed to it.

Even though using untyped parameters is flexible, it's no longer considered appropriate since the introduction of generic types. *Generic types* allow you to dynamically assign the typing for data objects. You'll learn more about generic types in Chapter 16 when we discuss dynamic programming.

The following is the syntax for typed parameters:

```
FORM abc using u1 type t.
ENDFORM.
```

With typed parameters, you define the data type statically. Typed parameters are considered more efficient than untyped parameters. You must use generic types instead of untyped parameters if the requirement is to determine the data type dynamically during the runtime. For this reason, we discourage you from using untyped parameters.

The following points apply when working with typed formal parameters:

- The syntax of the `FORM` statement is a bit restricted, so you can only specify a data type on the `FORM` statement and can't use the `length` addition when using elementary types. For this reason, we recommend typing the formal parameters by referring them to either user-defined global types in the program or data types from the ABAP Data Dictionary.
- The type and length of the actual parameter should match the formal parameter. You usually don't pass one data type to a different data type.
- Passing a variable of the wrong data type or length to a typed parameter causes a syntax error.
- Typed formal parameters are more efficient because the system doesn't need to determine the data type at runtime to allocate memory.
- They help prevent coding errors because the syntax checker will prevent you from passing incompatible parameters.
- They also help prevent runtime errors. If you perform an arithmetic operation on an untyped formal parameter, and a character value is passed to it, it will result in a runtime error.

- You always use typed parameters when passing structures. If a structure is passed to an untyped formal parameter, you can't access the fields of the structure statically because the system will identify the formal parameter as a structure only at runtime.

In the following subsection, we'll discuss how to pass parameters and internal tables to subroutines.

Passing Parameters to Subroutines

There are three different ways that parameters can be passed to subroutines:

- **Pass by reference**
  Parameters passed by reference.
- **Pass by value**
  Input parameters that pass values.
- **Pass by value and result**
  Output parameters that pass values.

The `USING` and `CHANGING` additions for the `FORM` statement define the parameter interface and how the parameters are passed. The same additions are also used for the `PERFORM` statement to pass the actual parameters, but purely for documentation purposes; they don't influence how the parameters are passed in that case.

The sequence of the additions is fixed. `USING` comes first, followed by `CHANGING`. You can use either one of the additions or both of them in the statement. Each addition can be followed by a list of any number of parameters.

The syntax at the `PERFORM` level is as follows:

```
PERFORM abc USING f1 f2 CHANGING f3
```

The parameters are listed under either `USING` or `CHANGING`, depending on whether the value of the actual parameter is changed after the subroutine call or remains unchanged. If the actual parameter is changed after the subroutine call, list it under `CHANGING`; otherwise, list it under `USING`.

Listing 7.16 shows an example of the `USING` addition.

```
DATA f1 TYPE c VALUE 'A'.
PERFORM abc USING f1.
FORM abc USING p1.
```

```
WRITE p1.
ENDFORM.
```

Listing 7.16 Passing Parameters with the USING Addition

In Listing 7.16, actual parameter `f1` is passed to formal parameter `p1` in the subroutine `abc`. Because the value of actual parameter `f1` doesn't change after the subroutine call, we used the `USING` addition at the `PERFORM` statement.

In Listing 7.17, the value of actual parameter `f1` will change from `A` to `B`, so we used the `CHANGING` addition at the `PERFORM` statement. Note that the `USING` or `CHANGING` additions are only to help other developers looking at the code to identify if the actual parameter simply is being used or actually is changed within the subroutine.

```
DATA f1 TYPE c VALUE 'A'.
PERFORM abc CHANGING f1.
FORM abc USING p1.
P1 = 'B'.
ENDFORM.
```

Listing 7.17 Passing Parameters with the CHANGING Addition

The example in Listing 7.15 will produce the same result and behave the same way even if we use the `CHANGING` addition at the `PERFORM` statement or the `USING` addition as in Listing 7.16.

In the following subsections, we'll introduce the different methods for passing parameters to subroutines.

Parameters Passed by Reference

The syntax at the `FORM` statement determines how the parameters are passed. When the parameters are passed by reference, the formal parameter points to the same memory location as the actual parameter. In other words, a new memory isn't created for the formal parameter; instead, a pointer is created that directs to the memory location of the actual parameter. This way of passing the parameters is efficient (especially for internal tables) because no copy of the data is made, and no extra memory is used.

When parameters are defined with just the `USING` or `CHANGING` additions at the `FORM` statement, they're passed by reference by default. Some examples include the following:

- `FORM abc USING p1.`
  `ENDFORM.`
- `FORM abc CHANGING p1.`
  `ENDFORM.`
- `FORM abc USING p1 CHANGING p2.`
  `ENDFORM.`

When the parameters are passed by reference, the value of the actual parameter changes immediately the moment the contents of the formal parameter are changed because both point to the same memory location. You need to be careful while programming because the original data is instantly changed. If any conditions aren't met later in the program logic and you want to quit the subroutine, changes to the original data can't be reverted.

When passing the parameters by reference, `USING` and `CHANGING` are equivalent, but for documentation purposes, you should use `USING` for input parameters (which aren't changed in the subroutine) and use `CHANGING` for output parameters (which are changed in the subroutine).

If you don't want the value of an actual parameter to be changed automatically, you must pass it by value.

Input Parameters That Pass Values

Parameters are passed by value when the `VALUE` addition is used after the `USING` addition, as shown in Listing 7.18.

```
FORM abc USING VALUE (p1)
               VALUE (p2).
ENDFORM.
FORM xyz USING VALUE (p1) CHANGING p2.
ENDFORM.
```

Listing 7.18 Passing by Value

In Listing 7.18, the parameters `p1` and `p2` are passed by value in subroutine `abc`, whereas in subroutine `xyz`, the parameter `p1` is passed by value and `p2` is passed by reference. You use the `USING` addition only once in the syntax and precede each parameter that needs to be passed by value with the `VALUE` addition.

When the parameters are passed by value, the actual parameter and the formal parameter occupy their own memory space. When you call the subroutine, the value of the

actual parameter is passed to the formal parameter (it creates a copy of the original data). If the value of the formal parameter changes, this has no effect on the actual parameter. Passing parameters by value is to be avoided, especially when passing large internal tables. However, passing by value allows you to work with a copy of the original data without worrying about changing the original data because the value of the actual parameter is never changed. Actual parameters are used for input parameters that the subroutine receives.

Output Parameters That Pass Values

If you want to work with a copy of the output parameters, you can pass them by using the `VALUE` addition with the `CHANGING` statement. This way, a new memory is created for the formal parameter, and the contents of the actual parameter are copied to the formal parameter (similar to the input parameters passed by value).

However, the difference is that the actual parameter will be changed after the subroutine call. The contents of the formal parameter are copied to the actual parameter either when `ENDFORM` is executed or when the subroutine is terminated with the `CHECK` or `EXIT` statements.

The actual parameter remains the same if the subroutine is terminated prematurely with an error message. This allows you to change the actual parameter only when all the conditions are met, unlike pass by reference cases, in which the actual parameter is changed instantly.

The syntax to pass parameters by value and result is shown in Listing 7.19.

```
FORM abc CHANGING VALUE (p1)
                  VALUE (p2).
ENDFORM.
FORM xyz USING VALUE (p1)
         CHANGING VALUE (p2).
ENDFORM.
```

Listing 7.19 Passing by Value and Result

In Listing 7.19, the parameters `p1` and `p2` of subroutine `abc` are passed by value, and both are output parameters. In subroutine `xyz`, the parameter `p1` is an input parameter that is passed by value, and `p2` is an output parameter that is passed by value. You use the `CHANGING` addition only once in the syntax and precede each parameter that needs to be passed with the `VALUE` addition.

Passing Internal Tables to Subroutines

Passing fields to subroutines is simple, but there are a few things to keep in mind when passing internal tables. Internal tables can be passed a couple of ways: USING or CHANGING statements and the TABLES keyword, for example:

```
FORM abc TABLES pt_sflight.
ENDFORM.
```

Using the TABLES keyword allows you to pass both internal tables and their header lines (if the actual parameter is an internal table with header line). However, using the TABLES keyword is obsolete, and we don't recommend it.

Always pass internal tables with the USING or CHANGING statements. In addition, always use typed parameters when passing internal tables. The syntax checker will throw an error if any internal table–specific statements, such as LOOP or READ, are used statically on the formal parameter. This is because there is no way for the syntax checker to recognize the formal parameter as an internal table when using untyped formal parameters.

To type the formal parameter, you can refer the formal parameter to a structure or table type. The syntax to type an internal table is similar to that for the DATA keyword. When referring to a structure, you can use the syntax TYPE TABLE OF, as shown in Listing 7.20.

```
TYPES: BEGIN OF ty_makt,
         matnr type matnr,
       END OF ty_makt.
FORM abc USING pt_makt TYPE STANDARD TABLE OF ty_makt.
ENDFORM.
```

Listing 7.20 Defining an Internal Table as a Formal Parameter

Table types are the data types for internal tables. Similar to how a structure type defines a structure, a table type defines an internal table. When a data object is referred to a table type using the TYPE addition, it creates an internal table. Table types can be defined either locally in ABAP programs or globally in the ABAP Data Dictionary. To define a table type locally, use the OCCURS clause with the TYPES keyword, as shown in Listing 7.21.

```
TYPES: BEGIN OF ty_makt,
         matnr type matnr,
       END OF ty_makt.
```

```
TYPES: tbty_makt TYPE ty_makt OCCURS 0.
FORM abc USING pt_makt TYPE tbty_makt.
ENDFORM.
```

Listing 7.21 Defining an Internal Table Using a Table Type

Table types are more useful in function modules and methods, where the typing is restricted to `TYPE` or `LIKE` statements and doesn't allow the use of the `TABLE OF` addition.

You can also type the formal parameter by referring to a global internal table via the `LIKE` addition (see Listing 7.22).

```
TYPES: BEGIN OF ty_makt,
         matnr type matnr,
       END OF ty_makt.
DATA: it_makt TYPE STANDARD TABLE OF ty_makt.
FORM abc USING pt_makt LIKE it_makt.
ENDFORM.
```

Listing 7.22 LIKE Addition

Table types can be defined globally in the ABAP Data Dictionary. To create a global table type, follow these steps:

1. Open Transaction SE11, and select the **Data type** radio button. Provide the name of the table type, and click **Create**. The name of the table type should be in a customer namespace, starting with Y or Z (as shown in Figure 7.7).

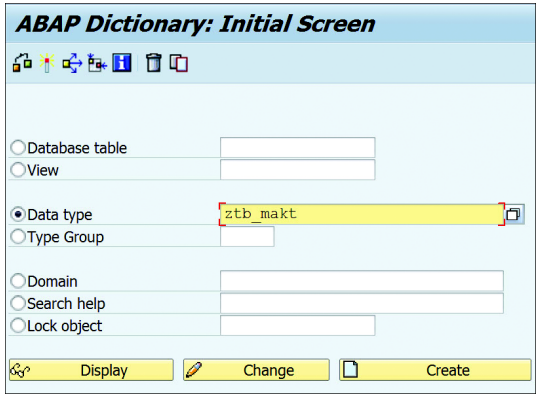


Figure 7.7 ABAP Data Dictionary: Initial Screen

2. In the modal dialog box that appears, select the **Table type** radio button and click **OK** (see Figure 7.8).

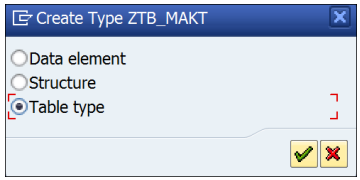


Figure 7.8 Modal Dialog Box to Select Table Type

3. On the next screen, enter the **Short text** field and the structure name or the ABAP Data Dictionary table name based on the **Line Type** you want the table type to contain, as shown in Figure 7.9. The line type mentioned here will form the components of the table type.

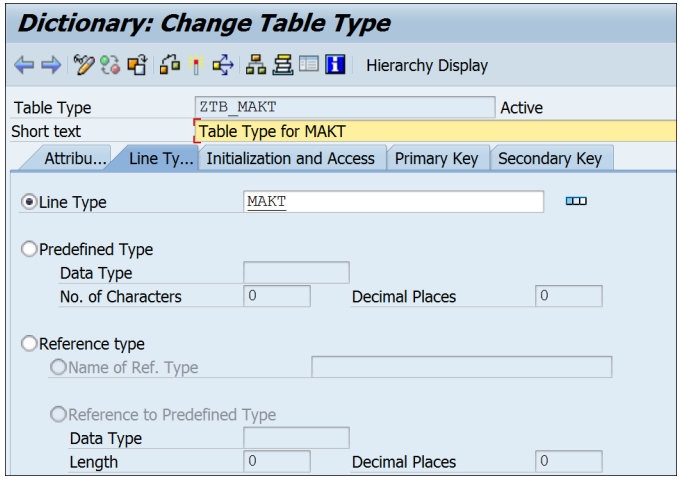


Figure 7.9 Maintaining Line Type for the Table Type

4. Click **Activate** to activate the table type.

The preceding steps quickly create a table type with default options. However, you can determine more properties for the table type using different tabs on the screen. We'll dive into this topic further in Chapter 10.

7.4.2 Function Modules

Function modules are defined in function groups but can be called from all ABAP programs. Function groups act as containers for function modules that logically belong together. You can create function groups and function modules in the ABAP Workbench using the Function Builder tool (Transaction SE37).

Function modules are managed in a central function library, making it easy to look for existing function modules. The SAP system comes with many standard function modules that can be called from any ABAP program. Function modules also support RFC and thus can be called from remote systems using RFC.

One type of function module, the *update module*, is useful when performing database updates. Update modules allow you to extend the scope of the logical unit of work (LUW) from a single dialog step to multiple dialog steps. An LUW is a unit that exists between the PAI of one screen and the PBO of the next screen. We'll discuss LUWs further in Chapter 11.

Unlike subroutines, function modules aren't defined in an ABAP program's source code. Instead, they're defined using the Function Builder. Because function modules are always called externally and provide generic functionality, raising messages in a function module to handle error situations doesn't make sense. If an error occurs, function modules support exception handling, which allows you to catch certain errors while the function module is running.

Function modules are called by many programs. Changing an existing function module will affect all the programs that call that particular function module. If the parameter interface is changed, then all the programs that call the function module should be modified to reflect the new interface. If the function module isn't changed, the programs will throw runtime errors.

To prevent this, the Function Builder has a release process for function modules. By releasing the function module, you can ensure that incompatible changes can't be made to the function module, particularly to the interface. Programs that use a released function module won't cease to work if the function module is changed.

Function modules are created in *function groups* that act as containers for function modules. Function groups can't be executed directly. Rather, they're loaded into memory when any function module of the function group is called. Note that all the function modules belonging to the function group are loaded into memory when any one of its function modules is called. For this reason, even though you can define

multiple function modules in a function group, we recommend keeping related function modules in the same group.

In this section, we'll discuss the steps for creating and calling function modules.

Creating Function Modules

The following steps walk you through creating a function module:

1. Open Transaction SE37 to access the Function Builder.
2. As shown in Figure 7.10, navigate to **Goto • Function Groups • Create Group** to create a new function group.

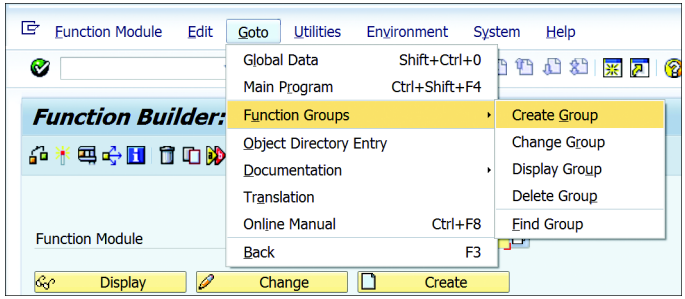


Figure 7.10 Menu Path to Create Function Groups Using the Function Builder

3. On the dialog screen that appears, as shown in Figure 7.11, enter a name for your function group (e.g., "zcb_fg"). The name of a function group can be up to 26 characters long. This name is used by the system to create the components of the group (the main program and corresponding include programs).

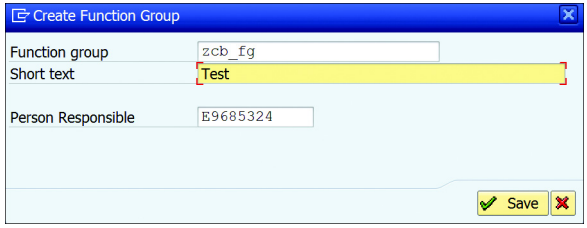


Figure 7.11 Create Function Group Screen

When you create a function group or function module in the Function Builder, the main program and include programs are generated automatically (see Figure 7.12).

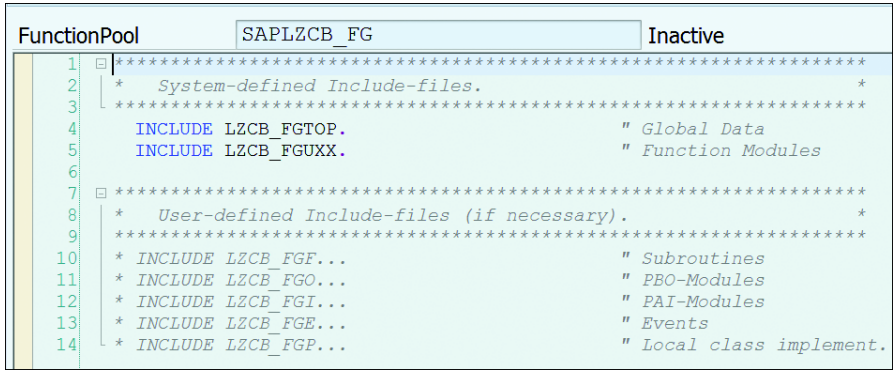


Figure 7.12 Function Group Includes

As shown in Figure 7.12, the main program SAPLZCB_FG contains nothing but the INCLUDE statements for the following include programs:

- **LZCB_FGTOP**: Contains the FUNCTION-POOL statement and global data declarations for the entire function group.
- **LZCB_FGUXX**: Contains further INCLUDE statements for the include programs LfgrpU01, LfgrpU02, and so on. These INCLUDE statements contain the actual function modules.
- **LfgrpF01** and **LfgrpF02**: Subroutines can be maintained in the INCLUDE programs LfgrpF01, LfgrpF02, and so on. They can be called from all function modules of the group as internal subroutines.

All the function modules in the function group can access the data declarations made in the global data area of the function group. For this reason, you should place all function modules that use the same data in a single function group instead of creating a new function group for each function module. For example, if you have a set of function modules that all use the same internal table, you could place them in a function group containing the table definition in its global data.

Function groups, like executable programs and module pools, can contain screens. Depending on the type of screen, user inputs can be processed either in dialog modules or in the corresponding event blocks in the main program of the function group.

Data declarations made within the function module between the FUNCTION…ENDFUNCTION statements are local to the specific function module. These data de-

clarations can be accessed only within the function module in which they are defined.

4. After the function group is created, you can create the function module within this function group. To do so, enter the name of the **Function Module** on the initial screen of the Function Builder, and click **Create** (Figure 7.13).

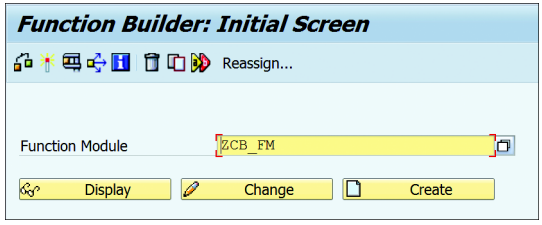


Figure 7.13 Creating Function Module

5. In the dialog box that appears, input the name of the **Function group** and a short description (**Short text**) (Figure 7.14). Click **Save**.

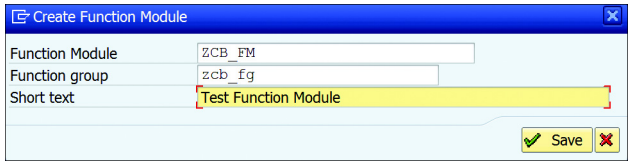


Figure 7.14 Assigning the Function Module to the Function Group

This will open the function module editor, as shown in Figure 7.15.

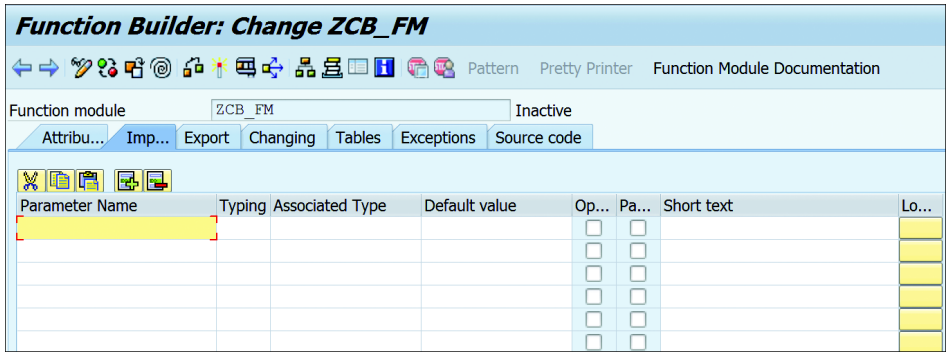


Figure 7.15 Function Module Editor

The editor contains seven tabs:

- **Attributes:** The most important setting in the **Attributes** tab is **Processing Type**. Here, you can choose to create a **Normal Function Module**, a **Remote-Enabled Module**, or an **Update Module**. By default, the **Normal Function Module** is selected (as shown in Figure 7.16).

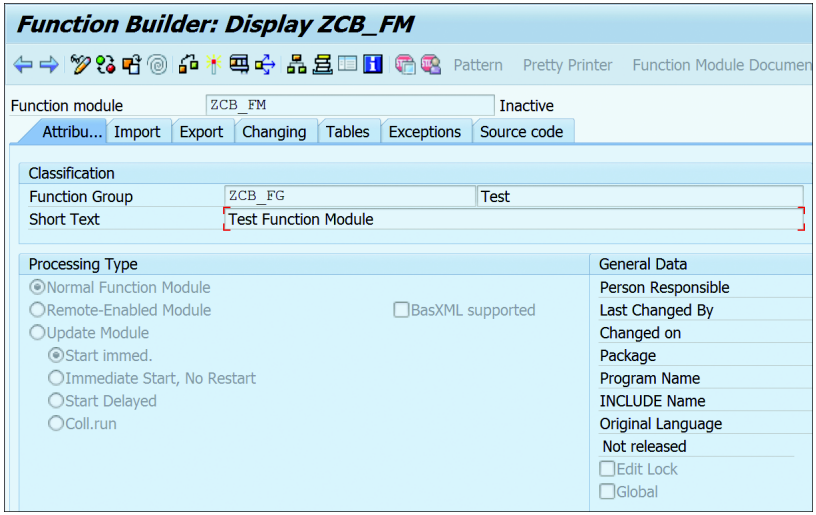


Figure 7.16 Attributes Tab

  Normal function modules can be called from any ABAP program in the SAP system. Remote-enabled function modules can be called from a remote system. These types of function modules help with interactions between SAP systems or with interactions between SAP systems and remote systems (through remote communications). Update modules are used during updating. We'll look at update techniques in Chapter 11.

- **Import:** In this tab, you define all the importing parameters that will be imported from the calling program. Import parameters can't be changed in the function module. As shown in Figure 7.17, in the **Parameter Name** column, you enter the name of your importing parameter. In the second column, **Typing**, you provide the type specification (either `TYPE` or `TYPE REF TO`). In the third column, **Associated Type**, you provide the reference to a data type defined in the ABAP Data Dictionary.

In the **Default Value** column, you can assign a default value to the parameter that's automatically filled when the function module is called in any ABAP program. The **Optional** column has a checkbox that can be ticked to make the parameter optional. Optional parameters can be ignored by the calling program. The **Pass Value** column checkbox allows you to pass the parameter by value. Otherwise, the parameter is passed by reference. You can maintain **Short text** and **Long Text** in the corresponding columns to describe the parameter for other developers, as shown in Figure 7.17.

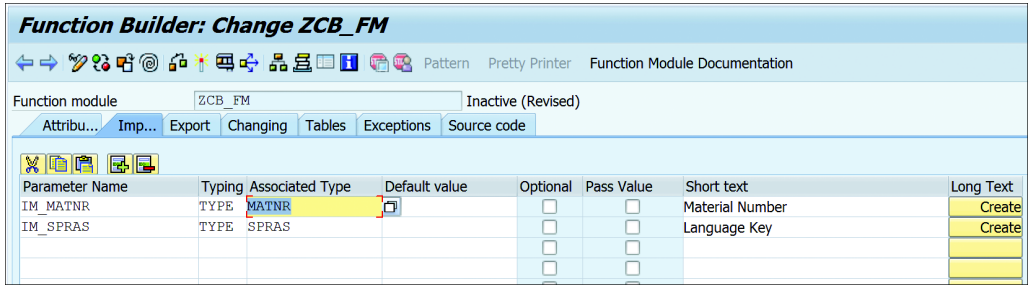


Figure 7.17 Import Tab

- **Export:** You define all the parameters that will be exported to the calling program under the **Export** tab. Except for the missing **Default Value** and **Optional** columns, as shown in the **Import** tab, all other columns are similar to the **Import** tab (Figure 7.18). These parameters are filled in the function module and sent to the calling program. The program can't send any values to the function module using these parameters.

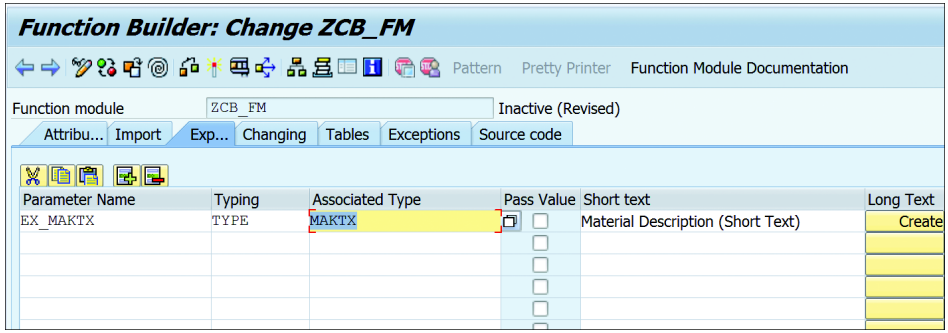


Figure 7.18 Export Tab

- **Changing**: In this tab, you define `CHANGING` parameters. `CHANGING` parameters work as both importing and exporting parameters (i.e., the function module can pass values to and import values using these parameters). The columns are identical to those in the **Import** tab.
- **Tables:** This tab is used to define internal tables. However, it's considered obsolete, and it's recommended to define internal tables in the **Import**, **Export**, or **Changing** tabs per requirements. You refer the parameter to a structure or database table using the `LIKE` specification to define an internal table in the **Tables** tab. This creates an internal table with a header line. To define an internal table in the **Import**, **Export**, or **Changing** tabs, use the `TYPE` specification and refer the parameter to a table type.
- **Exceptions:** In this tab, you can maintain exceptions that can be raised in the function modules. When an exception is raised, the function module is exited without executing further code. At this point, the control will return to the calling program, and the system field `sy-subrc` will be set to the exception number assigned while calling the function module. The value of the `sy-subrc` field can be checked in the calling program to implement suitable error handling, as shown ahead at line numbers 26 to 31 in Figure 7.22.

  You raise exceptions using the `RAISE` keyword in the source code of the function module, as shown ahead at line number 15 in Figure 7.20. To maintain the exception, enter the name of the exception in the **Exception** column and short and long text in the respective columns to provide additional information (Figure 7.19).

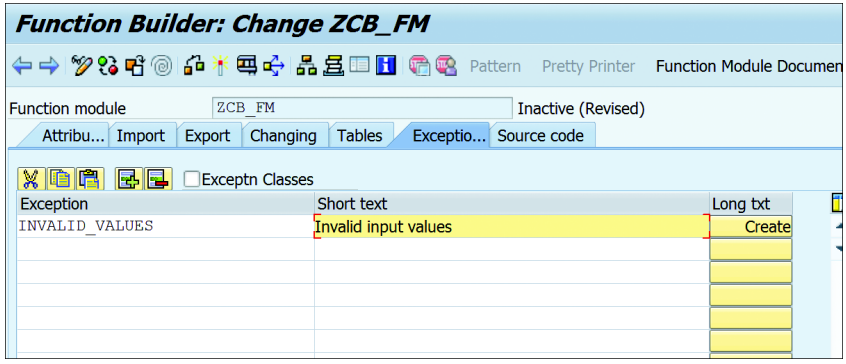


Figure 7.19 Exceptions Tab

- **Source code:** In this tab, you write the ABAP code for the function module within the `FUNCTION...ENDFUNCTION` block. Figure 7.20 shows code in which a function module is importing the material number and language through the `im_matnr` and `im_spras` parameters, respectively, and exporting the material description in the `ex_maktx` export parameter. If the `SELECT` statement fails, an exception is raised.

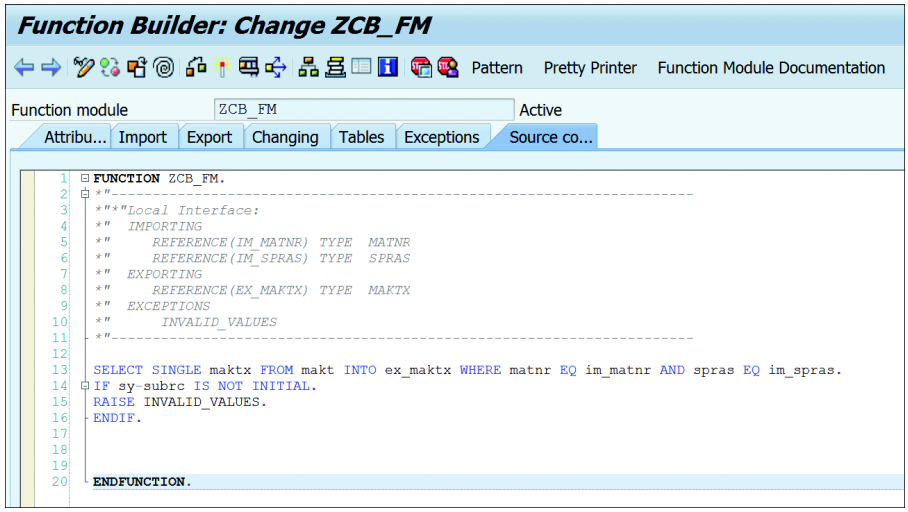


Figure 7.20 Source Code Tab

After the source code is maintained, activate the function module by clicking the **Activate** button. Make sure to select all inactive objects when the system prompts you to do so; the function group we first created wasn't activated separately.

Calling Function Modules

Function modules are called using the `CALL FUNCTION` statement in any ABAP program. To call a function module, you need to fill the parameter interface in the calling program. You can use the **Pattern** button in the application toolbar of the ABAP Editor to make this task easy (as shown in Figure 7.21).

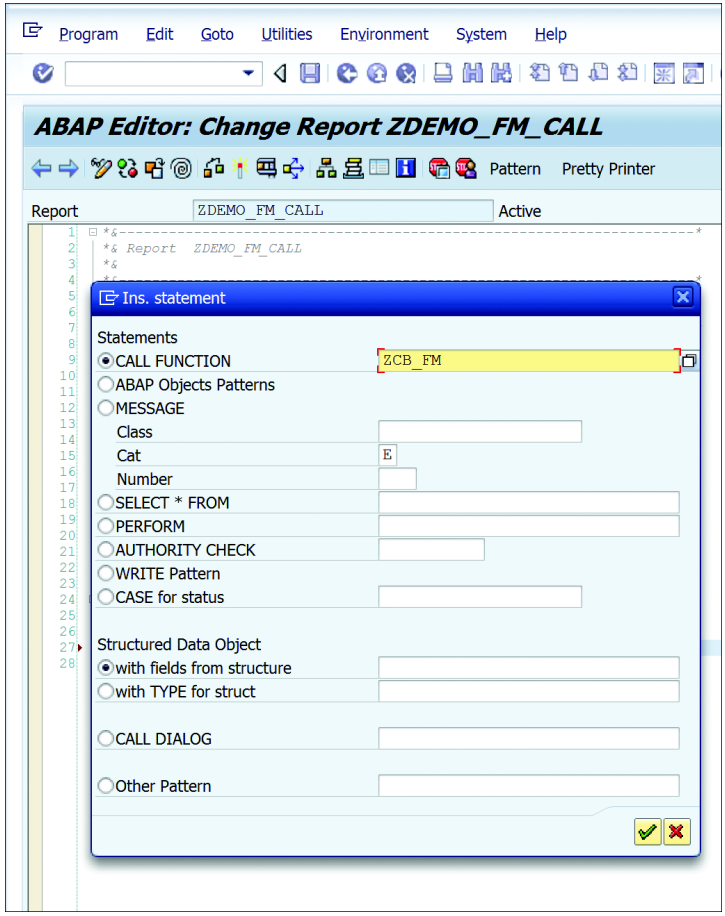


Figure 7.21 Using the Pattern Button to Call the Function Module

In the code generated by the **Pattern** button, simply map your actual parameters to the corresponding formal parameters, as shown in Figure 7.22. Always ensure that you've defined your actual parameters as the same type as the formal parameters. Passing incorrect or incompatible parameters results in runtime errors when the function module is called.

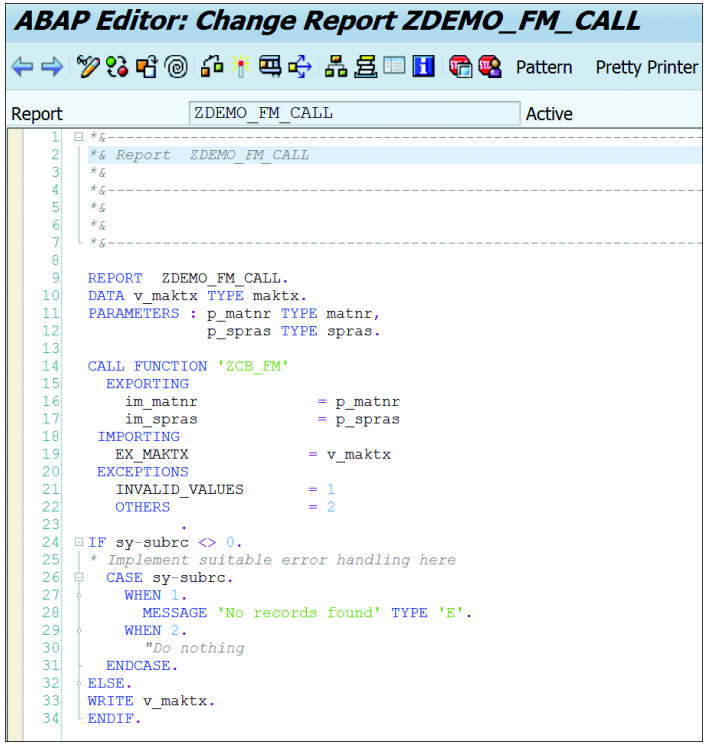


Figure 7.22 Function Module Call

7.4.3 Methods

Methods have been a part of procedures since ABAP Objects was first introduced. In this section, you'll learn how to work with methods and how to call methods into your ABAP programs. However, we'll have an in-depth discussion of OOP concepts and explore methods further in the next chapter.

Creating a Method

OOP takes a different approach than procedural programming. In procedural programming, you create procedures that take some input, process it, and export the results. In OOP, you deal with the objects you want to manipulate rather than the logic required to manipulate them. This approach helps developers identify the objects that require manipulation and establish a relationship between them. This process is called *data modeling*.

After an object is identified, a class is defined representing the object. Methods are then defined as tasks that can be performed on the object. Methods are the procedures in OOP, and variables are the attributes.

A *class* is like a generic representation of an object. In other words, an object exists as an instance of a class. For example, an employee class can be defined with methods and attributes—such as leaves, personal information, or compensation—that perform various actions. In addition, an object named James can be defined as an instance of the class employee.

You can define classes either globally in the repository or locally in an ABAP program. Global classes are created in class pool types of programs using the Class Builder (Transaction SE24).

Like function groups, you can't execute class pools directly. Each class pool contains the definition of one class. Objects (runtime instances) are created using the `CREATE OBJECT` statement in ABAP programs. These objects execute the statements of the class pool.

In this section, you'll learn how to create a global class and how to define methods within it. Like the function module example, we'll create a method that imports a material number and language and exports a description.

Creating a Global Class

To create a global class and define the methods within, follow these steps:

1. Open the Class Builder tool using Transaction SE24.
2. In the **Object type** field, enter the name of the class you want to create, and click the **Create** button. The name of the class should be in a customer namespace, starting with a Y or Z. We recommend starting the class name with the naming convention `ZCL_`; we'll discuss why shortly. Figure 7.23 shows the initial screen of the Class Builder.

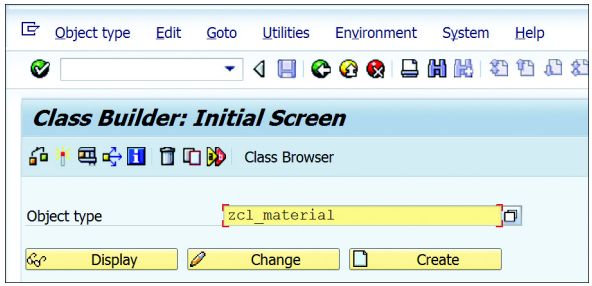


Figure 7.23 Class Builder: Initial Screen

The Class Builder is used to the create the class pool *and* the interface pool programs. If the first three letters of your object name are "zcl", then the Class Builder will create a class directly. Similarly, if the first three letters are "zif", then it will create an interface pool. However, if you start your object name with any other letters, the builder will prompt you to select whether you want to create a class pool or interface pool.

Interface pools don't contain any executable statements. Instead, they function as containers for interface definitions. When you implement an interface in a class, the interface definition is implicitly included in the class definition.

3. In the dialog box that appears, maintain a short description for your class, and leave other options set to their default values; we'll explore these other options in the next chapter. Click **Save,** as shown in Figure 7.24.

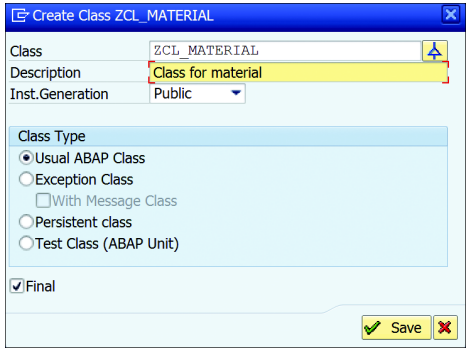


Figure 7.24 Maintaining Class Properties

4. This will open the form-based editor, with the **Methods** tab selected by default, as shown in Figure 7.25. The editor has eight tabs that allow you to maintain different properties. For now, use the **Methods** tab to create a method; we'll explore the other tabs in the next chapter after we discuss the basics of OOP.

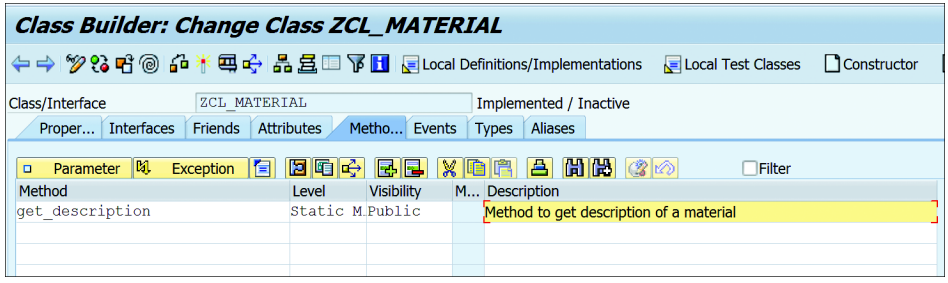


Figure 7.25 Form-Based Editor

The table control under the **Methods** tab has five columns:

- **Method**: This is where you create the name of your method.
- **Level**: Here, you maintain whether the method is a static or instance method. Static methods are similar to function modules in that one instance exists in the program when the method is called, whereas instance methods can have multiple instances in the same program. For now, create a static method. You can read more about instance methods in the next chapter. Use the F4 help to select the level.
- **Visibility:** In a class, attributes and methods can be set to a visibility level to encapsulate the implementation from the outside world. There are three visibility levels:
  - **Public**: Allows the attributes and methods to be accessed directly from any ABAP program.
  - **Protected**: Allows the attributes and methods to be accessed only from the methods of the same class or its subclasses.
  - **Private**: Allows the attributes and methods to be accessed only from the methods of the same class.

  Setting the right visibility level at the design stage to encapsulate the implementation is a great way to reduce bugs in your applications. For now, set the visibility of the method to **Public** because you'll be calling it in the program directly. We'll discuss encapsulation and implementation hiding in the next chapter.
- **Method type**: This is a non-editable field that displays the icon to highlight constructor method.
- **Description**: Maintain a short description for your method so other programmers can understand its purpose.

5. Click **Save** ❶ in the standard toolbar to save your changes, and then click the **Parameter** button ❷ above the **Method** column (see Figure 7.26) to maintain the parameter interface for the method. Make sure the cursor is on the method name when you click the **Parameter** button.
6. On the parameter interface screen, as shown in Figure 7.27, you can maintain importing, exporting, changing, and returning parameters. Importing, exporting, and changing parameters work like the parameters in a function module. Returning parameters are useful for special methods called functional methods, which just return one value, such as validation logic in which the method just returns a Boolean value: true or false. When you use returning parameters, you can't have any exporting parameters for the method.

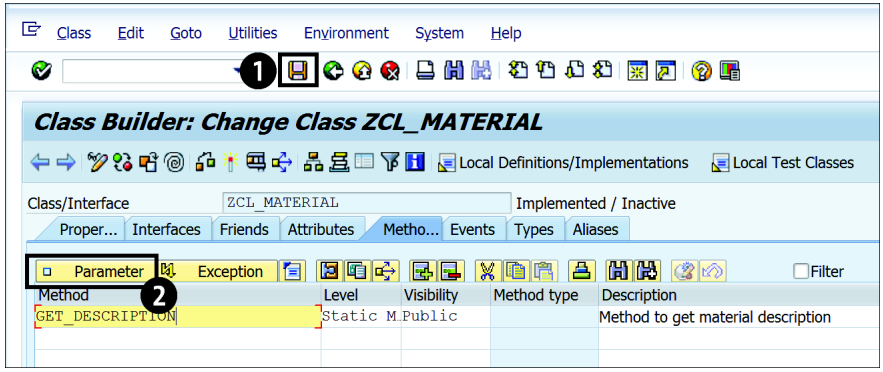


Figure 7.26 Save and Maintain Parameter Interface

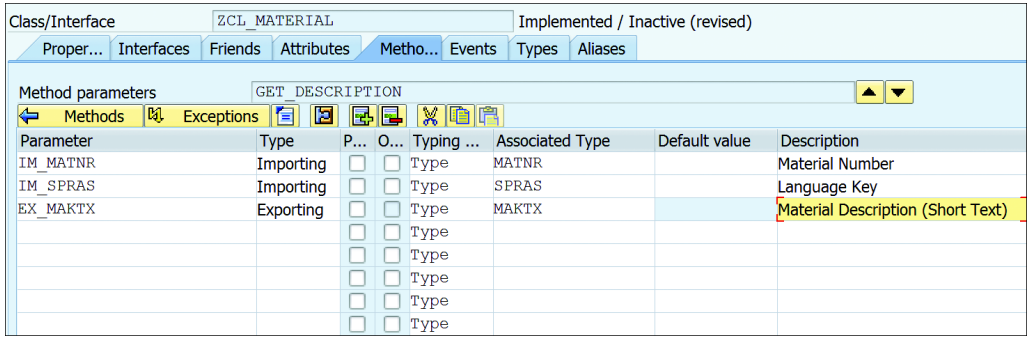


Figure 7.27 Maintain Parameter Interface

7. As shown in Figure 7.27, enter the name for your parameter, and use the F4 help to maintain the parameter type (either **Importing** or **Exporting** in this example).
8. In the **Associated Type** column, enter the data element reference for each parameter, as you did for function modules.
9. Maintain the description for the parameter in the **Description** column, or simply press Enter to pick the description associated with the data element from the ABAP Data Dictionary.
10. Click the **Activate** button in the application toolbar to activate your class and the method.
11. Once activated, click the **Code** button, as shown in Figure 7.28.

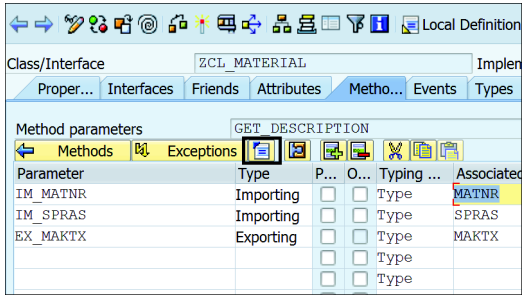


Figure 7.28 Click the Code Button to Maintain the Method Code

12. This will open the editor in which you can write the code between `METHOD` and `ENDMETHOD`, as shown in Figure 7.29. You can also click the **Signature** button to see the parameter interface. Click the **Activate** button to save and activate your changes.
13. Click the **Back** button twice to go back to the main screen.

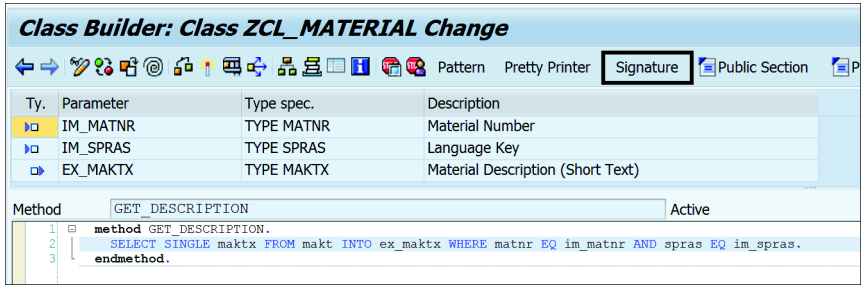


Figure 7.29 Writing Code in Method

Now that the method is implemented and maintained as a public method, it can be called from any program.

Calling a Method

Calling a static method is similar to calling a function module; it can be called directly using the `CALL METHOD` statement. You can use the **Pattern** button in the ABAP Editor to insert the statement, but to call an instance method, you first need to create a reference object, instantiate it, and then access the method using the reference object. We'll discuss instance methods in the next chapter.

Because we've created a static method in this example, use the **Pattern** button in the ABAP Editor, as shown in Figure 7.30.

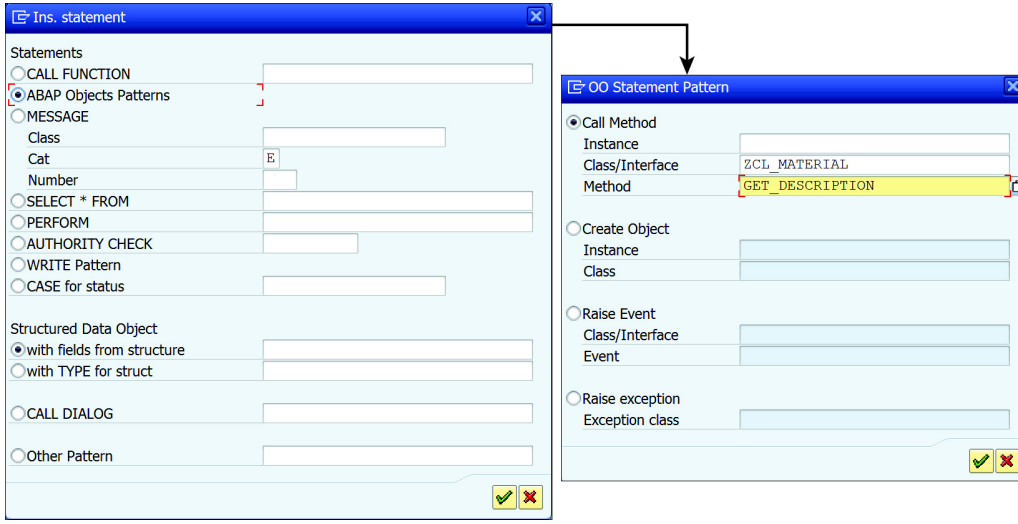


Figure 7.30 Calling a Static Method

Select the second radio button, **ABAP Objects Patterns**, and then the **Call Method** radio button on the next screen. Enter the class name and method name, and click **OK** to insert the statement.

The syntax used to call a static method is `CALL METHOD class_name=>method_name`, as shown in Figure 7.31.

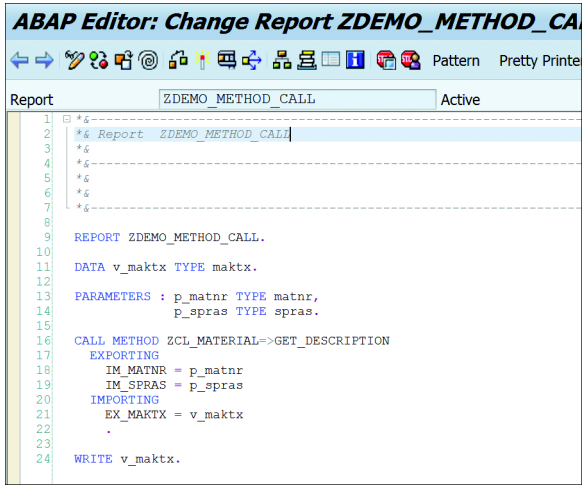


Figure 7.31 Calling Method in an ABAP Program

7.5 Inline Declarations

In SAP NetWeaver 7.4, SAP introduced inline declarations. *Inline declarations* allow you to declare data objects inline when required, as opposed to declaring them prior to their usage in an ABAP statement.

Most ABAP developers are taught to define data objects at the beginning of a program (for global data) or at the beginning of a procedure (for local data) and avoid using data declarations randomly throughout the code. However, many developers ignore these rules and declare the data object just before it's used in an ABAP statement, as it provides more clarity on where the data object is used. Many other programming languages (e.g., Hypertext Preprocessor [PHP]) don't require you to declare a data object before its use. With the introduction of inline declarations, you now have the flexibility to declare data objects when you need them as opposed to declaring them at the top of the program or procedure.

Even though it's possible to use inline declarations with a global scope (global data), it's advised not to use inline declarations for global data objects to avoid confusion. SAP recommends using inline declarations only within procedures to keep the scope of the data object local.

Inline declarations are very helpful when you have a long procedure with hundreds of lines of code, as it gets annoying to scroll up to the beginning of the procedure each time you want to define a new data object. Inline declarations are made using the keyword `DATA(..)` within the ABAP statement's operand position where the data object is expected in a writer position (i.e., where the data is written to a variable).

To get a better idea of how inline declarations have impacted ABAP coding, let's look at how to use inline declarations with examples comparing the old syntax. In the following subsections, we discuss some examples of when it's best to use inline declarations.

7.5.1 Assigning Values to Data Objects

In a pre-SAP NetWeaver 7.4 system, to assign a value to a data object, we first define the data object and then assign the value as shown:

```
DATA lv_book TYPE string.
lv_book = 'Complete ABAP'.
```

With SAP NetWeaver 7.4 and later, you can write the previous two lines of code as a single statement:

```
DATA(lv_book) = 'Complete ABAP'.
```

As you can see, inline declarations reduce the number of lines by half. When we use the statement DATA(..), the data object is automatically created when the code is compiled, irrespective of whether the statement is executed during runtime.

7.5.2 Using Inline Declarations with Table Work Areas

You can use inline declarations with table work areas as well. Listing 7.23 shows the pre-SAP NetWeaver 7.4 table work area.

```
DATA: lt_spfli TYPE STANDARD TABLE OF SPFLI,
      lw_spfli LIKE LINE OF lt_spfli.
LOOP AT lt_spfli INTO lw_spfli.
ENDLOOP.
```

Listing 7.23 Pre-SAP NetWeaver Table Work Area

Listing 7.24 shows the transformed SAP NetWeaver 7.4 or higher change with the inline declaration.

```
DATA: lt_spfli TYPE STANDARD TABLE OF SPFLI.
LOOP AT lt_spfli INTO DATA(lw_spfli).
ENDLOOP.
```

Listing 7.24 SAP NetWeaver 7.4 and Higher Table Work Area

You can use inline declaration for work areas with the READ statement as well:

```
READ TABLE lt_spfli INTO DATA(lw_spfli) INDEX 1.
```

7.5.3 Avoiding Helper Variables

Inline declarations are also useful when you want to avoid defining helper variables beforehand. For example, to find the number of rows in an internal table, we use the DESCRIBE TABLE statement in conjunction with a helper variable. A pre-SAP NetWeaver 7.4 approach would be as follows:

```
DATA lv_lines TYPE i. "helper variable
DESCRIBE TABLE lt_spfli LINES lv_lines.
```

With inline declarations, we can avoid declaring the helper variable `lv_lines` in the preceding statement, as shown:

```
DESCRIBE TABLE lt_spfli LINES DATA(lv_lines).
```

7.5.4 Declaring Actual Parameters

Another area where inline declarations are very useful is with the declaration of actual parameters. For example, when calling the method of a class, the actual parameters should match the *typing* of the formal parameters. To ensure the actual parameters are of the same type as the formal parameter, we check the signature of the method and define the actual parameters using the same typing as the formal parameters.

Navigating to check the signature of the method for declaring each actual parameter is annoying. In addition, each time the typing for the formal parameter is changed, the typing for the actual parameter in the program should also be changed. Inline declarations help you avoid all these problems as shown next.

Listing 7.25 shows how actual parameters were used before SAP NetWeaver 7.4.

```
DATA: lv_a1 TYPE string,
      lv_a2 TYPE string.
oref->meth( IMPORTING fp1 = lv_a1
                      fp2 = lv_a1 ).
```

Listing 7.25 Actual Parameters Pre-SAP NetWeaver 7.4

Listing 7.26 shows how we should use actual parameters after SAP NetWeaver 7.4.

```
oref->meth( IMPORTING fp1 = DATA(lv_a1)
                      fp2 = DATA(lv_a1) ).
```

Listing 7.26 Post-SAP NetWeaver 7.4 Actual Parameter Usage

As you can see, inline declarations can be used in any statement in the writer position. If used correctly, inline declarations can make your code leaner and easier to understand. As previously mentioned, use inline declarations only within procedures to limit their scope to the local data of the procedure.

7.6 Summary

In this chapter, we discussed various modularization techniques that can be used in ABAP programs. We began this chapter with an overview of key concepts and elements. You then learned about processing blocks and how they're called and implemented. You saw that event blocks are implemented by event keywords and called by the ABAP runtime environment.

Procedures are implemented by their own keywords and are called by specific keywords in the code. With this understanding of processing blocks, we're ready to discuss their usage when we cover some of the advanced topics in ABAP in upcoming chapters.

In the last section of this chapter, we looked at how inline declarations have simplified coding for ABAP developers. We did this by comparing a pre-SAP NetWeaver 7.4 line of code to an inline declaration-oriented code to highlight the significance of the change.

In the next chapter, we'll explore concepts related to OOP, and you'll learn more about classes and methods.