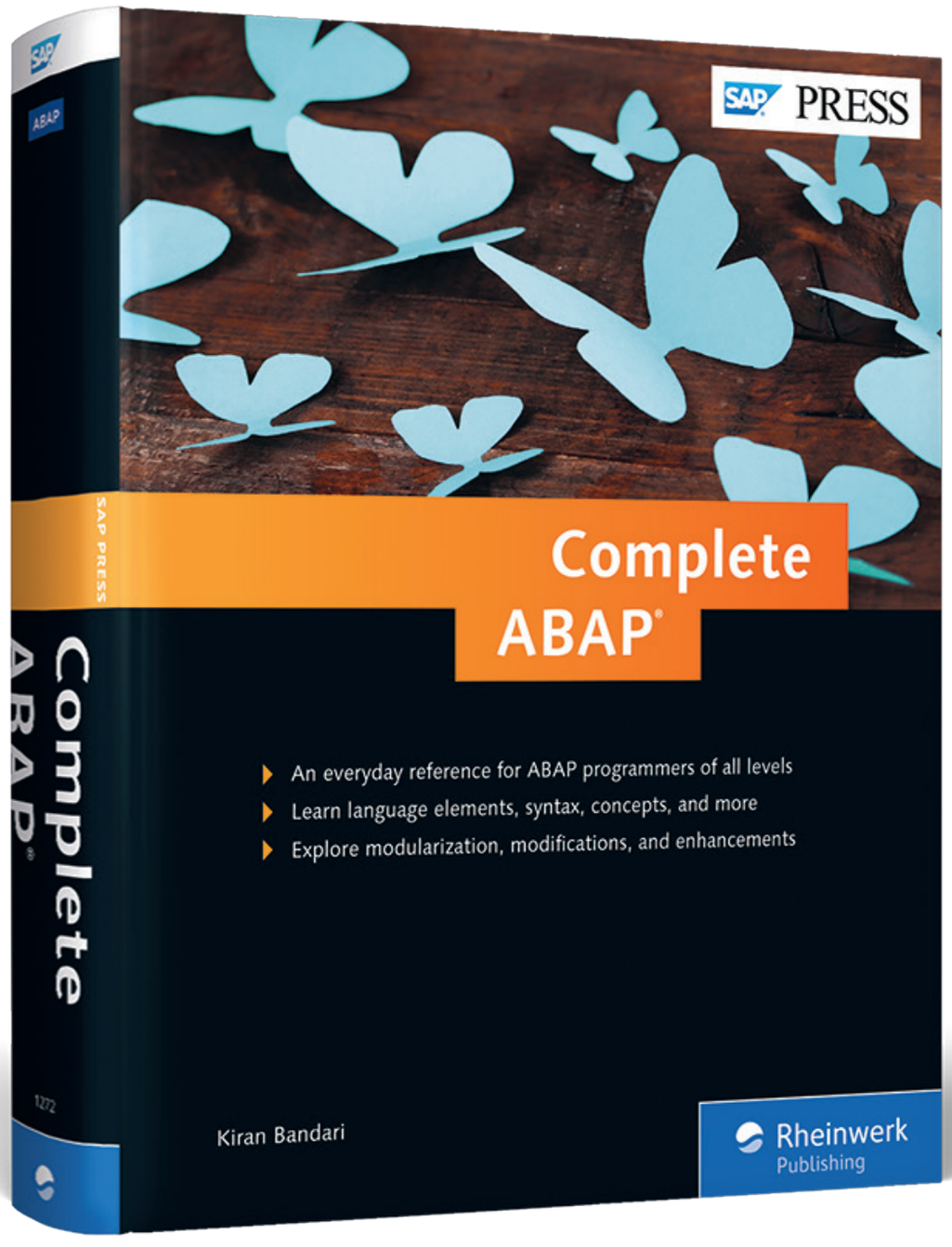


Reading Sample

In this reading sample, we provide two sample chapters. The first sample chapter introduces basic ABAP programming language concepts, which lay the foundation to writing ABAP programs. The second sample chapter discusses how to implement object-oriented programming techniques such as encapsulation, inheritance, and polymorphism in your code.

Kiran Bandari
Complete ABAP
1047 Pages, 2016, $79.95
ISBN 978-1-4932-1272-9

www.sap-press.com/3921

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

4 ABAP Programming Concepts

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, like user inputs on screens, data from a sequential file, or data from a database table, must always be transported to and saved in the program's local memory in order 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 once the program execution ends. If you wish to access this 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 in order to understand 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 in Chapter 8.

4.1 General Program Structure

Any ABAP program can be broadly divided into two parts:

- **Global declarations**
  In the global declaration area, global data for the program is defined; this data 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 which 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.

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.

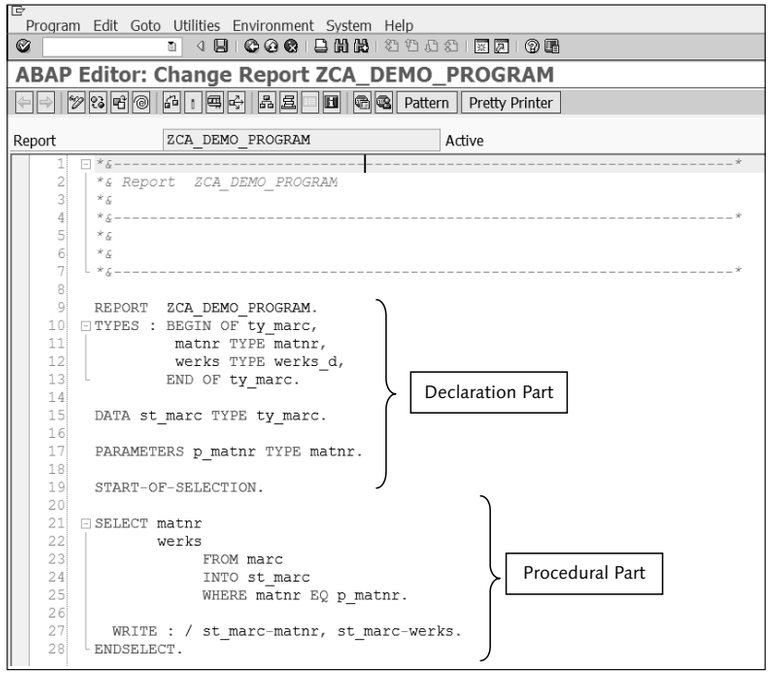


Figure 4.1 Program Structure

4.1.2 Procedural

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 and validate user inputs on the selection screen and 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, 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 work 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 (such as =, <, >), etc. 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 extended over several lines.
- ABAP code is not 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. Also, each ABAP word is separated by a space.

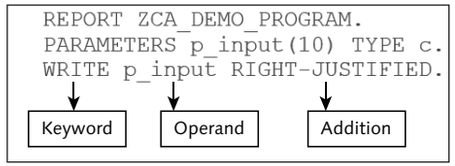


Figure 4.2 ABAP Statement

You can write multiple statements on one line or one statement can extend over multiple lines. Therefore, if you wish, 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 above 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 Example of Splitting 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 are called *chained statements*, and they help you avoid repeating the same keyword on each line.

For example,

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

Also can 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 are not 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 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 is not 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*, which 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 ABAP Editor. You can also visit *https://help.sap.com/abapdocu_750/en/abenabap.htm*, 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.

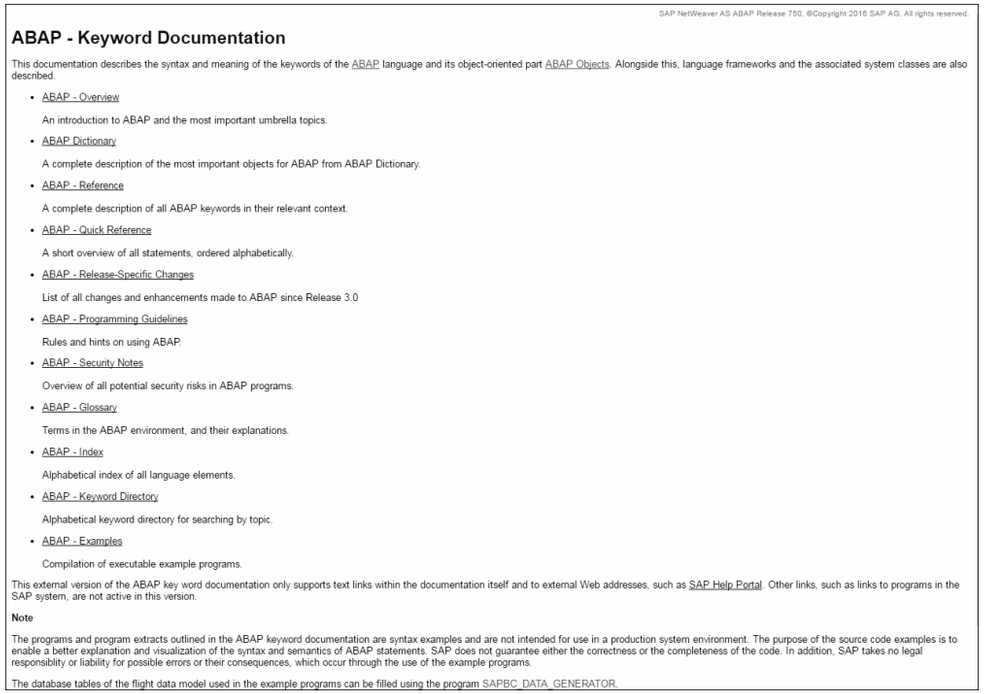


Figure 4.3 ABAP Keyword Documentation

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 to define data objects that store data in the program; these statements are called *data declarations*.

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. The customer's name consists of letters, a phone number consists of digits from 0 to 9, and the amount due 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`	Four-byte integer
`int8`	Eight-byte integer
`f`	Binary floating-point number
`p`	Packed number
`decfloat16`	Decimal floating-point number with sixteen decimal places
`decfloat34`	Decimal floating-point number with thirty-four 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 is not 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).

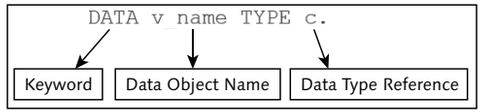


Figure 4.4 Declaring a Data Object

Notice that we did not 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 wish to have 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 five characters and a data object of `LENGTH 10` can store ten 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). 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 one byte for each character. For example, the letter A will occupy one byte in single-byte encoding.

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 seven-bit binary number.

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

- **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 four binary digits (bits), so a byte (eight 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, like 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—like Microsoft Excel—knows where each new field starts. `TYPE x` fields are also useful if you wish to create files in various formats from your internal table data.

Predefined Elementary ABAP Types with Variable Length

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 eighteen characters in length. Therefore, if you wish 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 is a character type with a variable length. It 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 like counters, indexes, time periods, and so on. Valid value range is -2147483648 to +2147483647. If the value range of `i` is too small for your need, 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 is not important. Example syntax: `DATA f1 TYPE f. "fixed length`

Table 4.3 Predefined Elementary Data Types and Use Cases

Data Type	Use Case

`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.`

`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 like 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 like 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; expected format is YYYYMMDD. Example syntax: `DATA f1 TYPE d.`
`t`	Use to process time; expected format is HHMMSS. Example syntax: `DATA f1 TYPE t.`

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

Data Type	Use Case

`x`	Use to process the binary value of the data. Useful for working with different code pages.
	Example syntax:
	`DATA f1 TYPE x.`
	`DATA f1(10) TYPE x.`
	`DATA f1 TYPE x LENGTH 10.`

`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 automatically convert the fields to numeric types 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`. Since `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 will 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 Listing 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. The following lists a few of these exceptions:

- A source field that only has blank characters is interpreted with the number 0.
- Scientific notations 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/abapdocu_750/en/abenconversion_elementary.htm*.

Even though the automatic type conversion makes it easy to move data between different types, do not 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 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 Leading 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 (like 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, it also allows your ABAP statements to interpret the data correctly and provide additional functionality.

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 does not 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 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 TYPE d Field

SAP allows you to set your own personal date format. So, say that User 1 has his personal date format set as MMDDYYYY, and User 2 has his personal date format set as DDMMYYYY. When 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 ABAP Data Dictionary.

Say 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 twenty characters. Later, if you want to extend the length of a person's name to thirty 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 the November 5, 2015, as the due date in MM/DD/YYYY format (11/05/2015), then the `SELECT` statement will fetch 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 own 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

Table 4.5 Dates Represented in Internal Format

Document_Number	Due_Date	Created_By
1003	20151105	USER2
1004	20151106	USER2

Table 4.5 Dates Represented in Internal Format (Cont.)

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 ten 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.

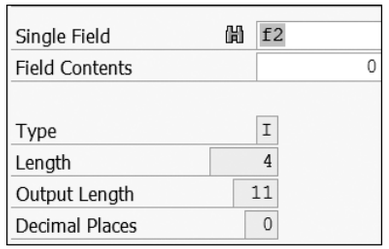


Figure 4.6 Output Length

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—eight bytes—as shown in Figure 4.7.

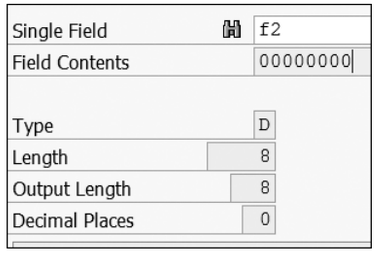


Figure 4.7 Output Length of Type d

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 ingnoring 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 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 ABAP Data Dictionary. As you may recall from Chapter 3, ABAP Data Dictionary is completely integrated into ABAP Workbench. This allows you to create and manage data definitions (metadata) centrally.

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

1. Open ABAP Data Dictionary via Transaction SE11 in the command bar or by navigating to the menu path Tools • ABAP Workbench • Development • ABAP Dictionary in SAP menu.

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).

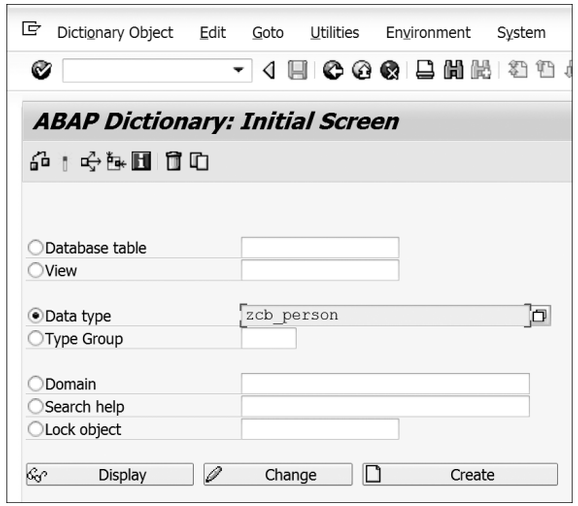


Figure 4.8 ABAP Data Dictionary: Initial Screen

3. The system presents a dialog box (see Figure 4.9) asking you to select the data type you want to create. 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.)

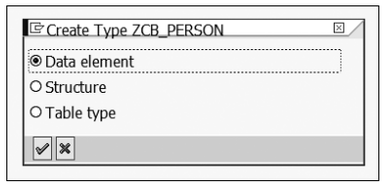


Figure 4.9 Create Type

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 selcted by default and the radio button ELEMENTARY TYPE preselected. Here, you have two options to maintain the technical attributes for your data element: You can either 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.

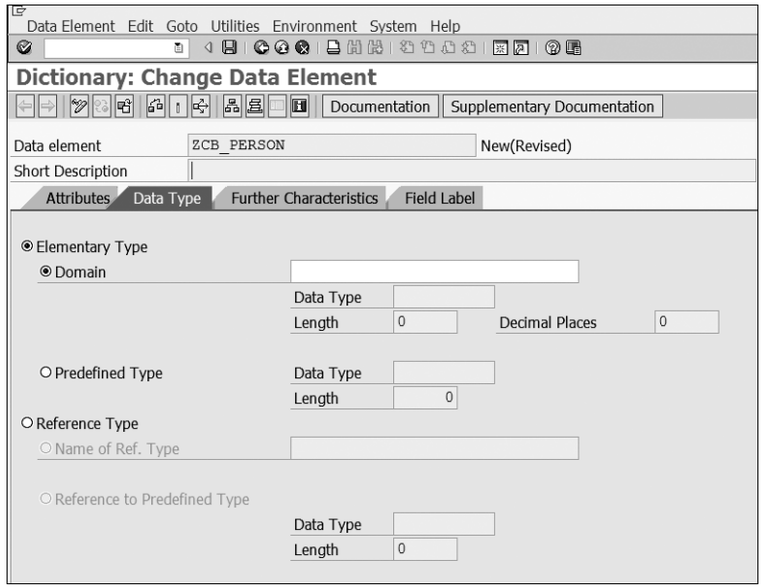


Figure 4.10 Change Data Element Screen

5. Select the PREDEFINED TYPE radio button and enter the predefined type name in the DATA TYPE field. Note that ABAP Data Dictionary has its own predefined elementary data types that correspond to the predefined elementary data types in 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).
7. Enter the length of the data type in the LENGTH field. For this data element, enter "20". 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.

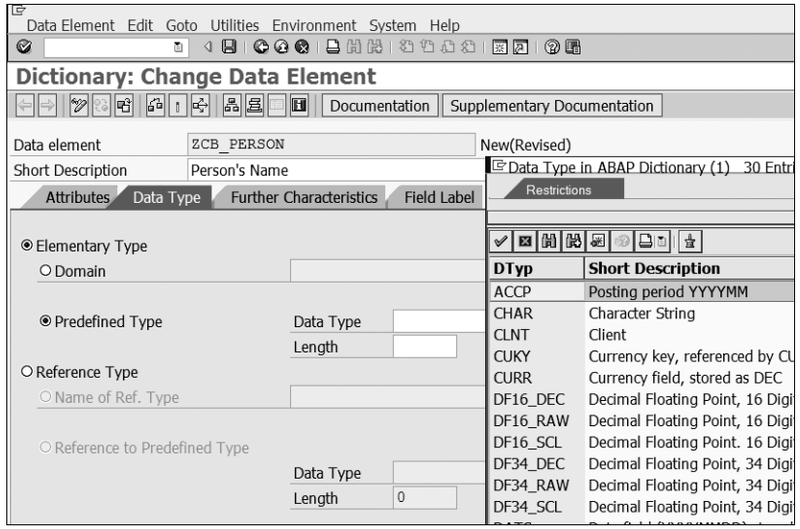


Figure 4.11 Data Type Value List

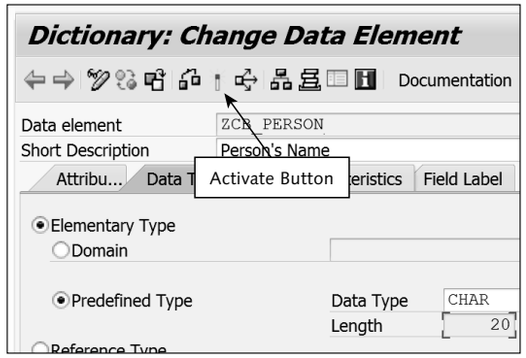


Figure 4.12 Activating a Data Element

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 refering to that data element. This allows you to centrally maintain the definitions of all related objects. 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, like search helps, field labels, and help documentation. The program fields can derive these properties automatically. We'll explore these concepts in greater 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, meaning that 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 ten-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.

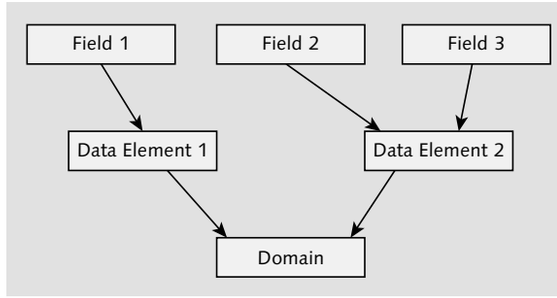


Figure 4.13 Fields, Data Elements, and Domain Relationships

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 *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 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 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.

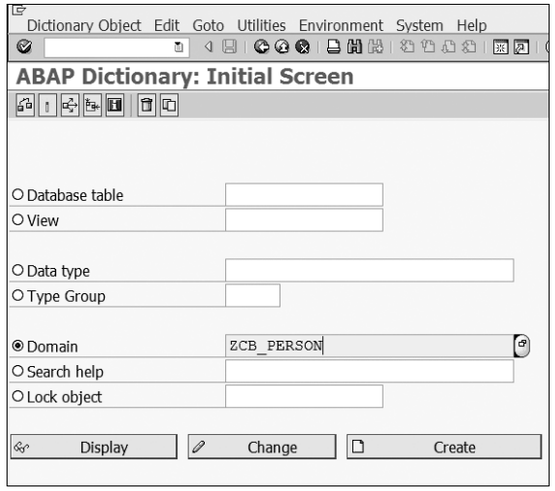


Figure 4.14 ABAP Dictionary: Initial Screen

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.

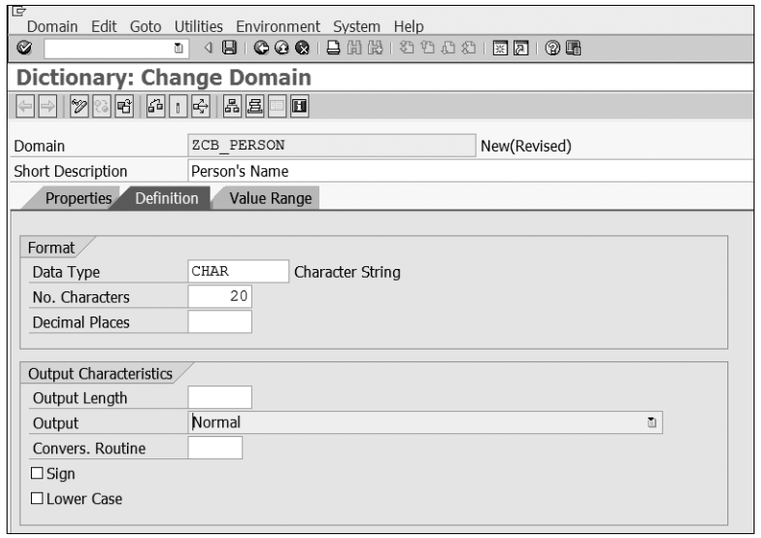


Figure 4.15 Change Domain Screen

4. You can also enter OUTPUT CHARACTERISTICS like output length, conversion routine, or disabling automatic uppercase conversion 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.
6. Notice that the DATA TYPE and LENGTH are chosen by the system automatically. If the system does not set the DATA TYPE and LENGTH after pressing the Enter key, it means the domain does not 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.

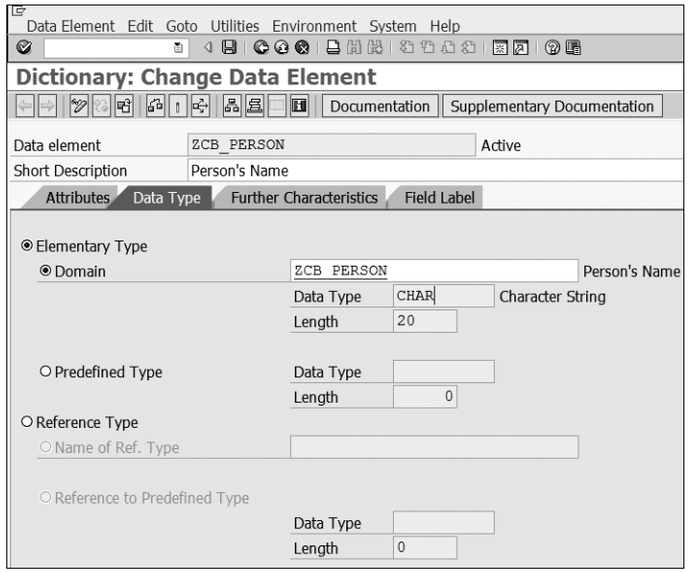


Figure 4.16 Attaching Domain to Data Element

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 are not 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 (such as 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 will look at the different classifications of data objects.

Literals

Literals are not 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 cannot access the memory of a literal to read its content, because it's an unnamed data object. This means that literals are not 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 to 9) that can have a prefixed sign. They do not 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, like `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 also 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 ten-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 cannot be changed by ABAP statements. The keyword used to declare a constant is `CONSTANT`. It is 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 is mandatory to use the addition `VALUE` to assign an initial value. This value cannot 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 is not 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 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 like 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 is beyond the scope of this book to cover all the available ABAP statements, so we'll give a brief introduction to

some popular statements here, and we'll explore these in greater detail in the 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 `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 will explore this topic further in Chapter 7.

  Examples include the `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 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, proceed with the following steps:

1. Open Transaction SE38 or follow the menu path 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.

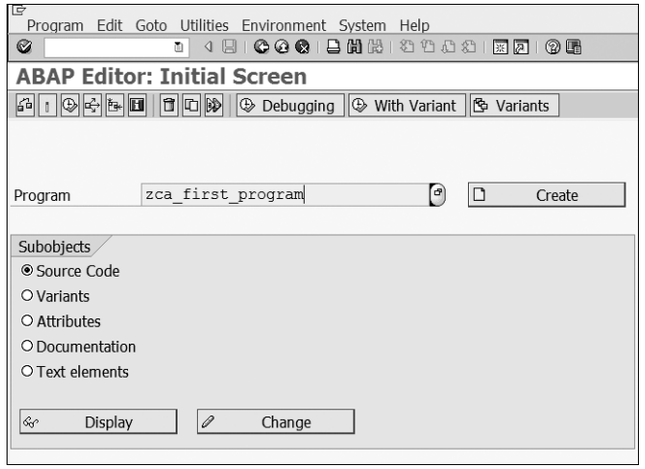


Figure 4.17 ABAP Editor: Initial Screen

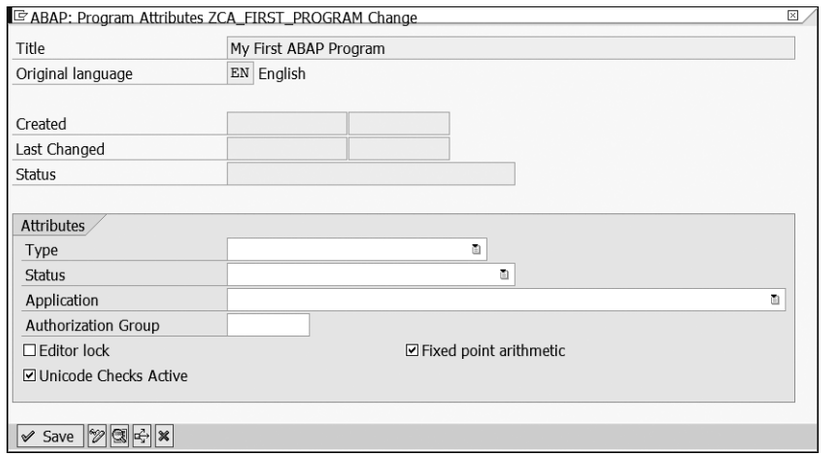


Figure 4.18 Program Attributes Screen

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 wish to create. This is the most important attribute and specifies how the program is executed. It's a mandatory field. From within ABAP Editor, you can only choose from the following program types: ▸ EXECUTABLE PROGRAM ▸ MODULE POOL ▸ SUBROUTINE POOL ▸ INCLUDE PROGRAM All other program types are not created directly from within ABAP Editor but are created with the help of special tools such as Function Builder for function groups or Class Builder for class pools.
STATUS	Allows you to set the status of the program development—for example, production program or test program.
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.
AUTHORIZATION GROUP	In this field, you can 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. 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. Allows you to specify the selection screen of the logical database that should be used.
EDITOR LOCK	If you set this attribute, other users can't change, rename, or delete your program. Only you will be able to change the program, its attributes, text elements, and documentation or release the lock.

Table 4.6 Program Attributes

Attribute	Explanation
Fixed Point Arithmetic	If this attribute is 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	This attribute allows you to set if the syntax checker should check for non-Unicode-compatible code and display a warning message. As of SAP NetWeaver 7.5, the system does not 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
   - Also check the Unicode Checks Active and Fixed point arithmetic checkboxes.

   Click the Save button when finished.

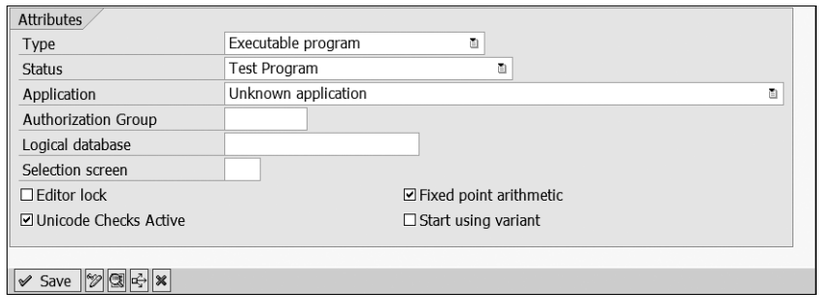


Figure 4.19 Maintaining Program Attributes

5. 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 of the ABAP Workbench objects assigned to one package are combined into one transport request.

   When working in 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.

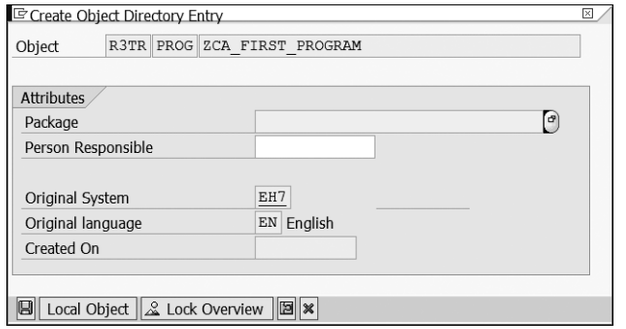


Figure 4.20 Assigning Package

6. Either enter "$TMP" as the Package and click the Save button or simply click the Local Object button (see Figure 4.20).
7. 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.
8. Write the following code in the program and activate the program:

```
PARAMETERS p_input TYPE c LENGTH 20.

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

9. 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 twenty 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, like you're doing with the WRITE statement to print it in the output.

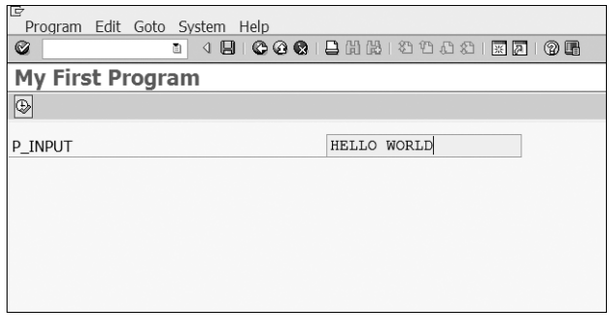


Figure 4.21 Selection screen

10. 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.

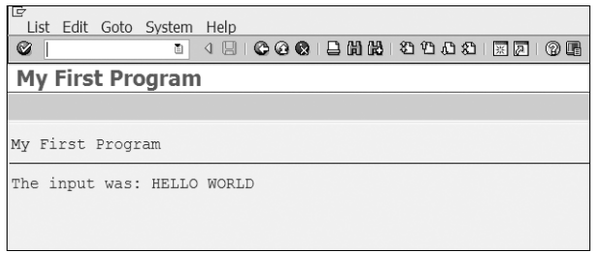


Figure 4.22 List Screen

11. 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.

We'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.

Some elements of ABAP programs can be designed using procedural or object-oriented programming. This chapter explores the concepts of object-oriented programming and their implementation in ABAP.

8 Object-Oriented ABAP

ABAP is a hybrid programming language that supports both procedural and object-oriented programming (OOP) techniques. In this chapter, we'll discuss the various concepts used in OOP and the advantages it provides over procedural programming techniques.

We'll start with a basic overview of OOP in Section 8.1. This introduction should help you appreciate the advantages of using OOP techniques. To understand OOP, you'll need to understand concepts such as encapsulation, inheritance, polymorphism, data encapsulation, and information hiding. We'll start examining those concepts in Section 8.2 with a look at encapsulation and the techniques to hide implementation from the outside world.

In Section 8.3, we'll discuss inheritance and the techniques that allow you to leverage existing features and functionality without having to reinvent the wheel. In Section 8.4, we'll look at polymorphism, which allows the same object to behave differently at runtime. We'll conclude the chapter with a discussion of XML in Section 8.5.

8.1 Introduction to Object-Oriented Programming

Before OOP, everything was based on functions and variables without focusing on the object itself. With OOP, the focus is on *objects*, which represent real-life entities, with functions and variables approximating the objects.

Traditional programming focused on developing programming logic to manipulate data—for example, a logical procedure that takes input data, processes it, and

produces output. However, OOP focuses on the object being manipulated rather than the logic to manipulate the data.

If you look around, you'll see many real-world objects, such as your car, your dog, your laptop, and so on. Each of these real-world objects is represented as a software object in OOP. Real-world objects have two characteristics: states and behaviors. For example, a car has states such as its current gear, current speed, and so on, and behaviors such as changing gears, applying brakes, and so on. Software objects that represent real-world objects also have states and behaviors; software objects store their states in variables (called attributes) and contain functions (called methods) to manipulate or expose the states of the object. For example, a car object can have attributes to store the current speed, current gear, and so on, and methods to read the current speed or change the gear.

Methods operate on the internal state of the object; that is, they can access the attributes of their own object and serve as a mechanism for communication between objects. Hiding this internal state of the object and routing all access to the object through its methods is known as *encapsulation*. Encapsulation is a fundamental principle of OOP, which we'll discuss in Section 8.2.

Sometimes, an object may share many similarities with other objects while having specific features of its own. For example, a mountain bike may have all the features of a road bike plus certain features unique only to mountain bikes. The concept of inheritance helps us leverage existing code and avoid code redundancy while extending functionality. We'll discuss inheritance in Section 8.3.

Identifying the states and behaviors of real-world objects is the first step to start thinking in terms of OOP. Before exploring OOP further, let's consider a simple procedural programming application that calculates a person's body mass index (BMI). We'll then look at how this application can be designed using OOP. For this example, we can create a function module with three interface parameters: height, weight, and BMI. We can pass the person's height and weight to the function and receive the calculated BMI as a result, as shown in Listing 8.1.

```
FUNCTION ZCALCULATE_BMI.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  IMPORTING
*"     REFERENCE(HEIGHT) TYPE I
*"     REFERENCE(WEIGHT) TYPE I
*"  EXPORTING
```



```
*"     REFERENCE(BMI) TYPE I
BMI = WEIGHT / HEIGHT.
ENDFUNCTION.
```

Listing 8.1 Example Function Module to Calculate BMI

Listing 8.2 shows the call to the function module `zcalculate_bmi` that we defined in Listing 8.1.

```
REPORT ZCA_BMI.
DATA : v_bmi TYPE I.
PARAMETERS: p_height TYPE I,
            P_weight TYPE I.

CALL FUNCTION 'ZCALCULATE_BMI'
EXPORTING
    height = p_height
    weight = p_weight
IMPORTING
    bmi = v_bmi.
WRITE: 'Your BMI is', v_bmi.
```

Listing 8.2 Program to Calculate BMI

When the function module is called in the program, the whole function group to which this function module belongs is loaded into the memory (internal session) the main program is assigned to. Each subsequent call to this function module in the program will access the existing instance in the memory as opposed to loading a new instance.

When a function module is called in a program, one instance of it is created in the internal session in which the program runs, and each subsequent call to this function module from the program will point to the existing instance that was previously loaded into the internal session. You can't have multiple instances of the same function module within the same internal session.

If you call the `zcalculate_bmi` function module in the program to calculate the BMI of multiple persons, there's no way for the function module to be aware of the person for which the BMI is calculated. You can define a global variable in the function group to store a person's name so that other function modules of the function group can use this information, but it's easy to lose track of all these variables in the code—and even then, this design can only support one person at a time in the application.

The focus of the function module here is to take a certain input and process it to produce a result. OOP shifts the focus to the object. In this example, we define an object that represents a person and then define attributes and methods to store and process the person's BMI.

In this section, we'll look at the defining elements of OOP, including classes, methods, instances and static components, and events.

8.1.1 Classes

Objects encapsulate data and create functions to manipulate that data. The characteristics (called *attributes*) and functions (called *methods*) of an object are put into one package. The definition of this package is called a *class*. To use objects in a program, classes must be defined, because an object exists as an instance of a class.

A class can be defined as shown in Listing 8.3.

```
REPORT ZCA_BMI.
CLASS CL_PERSON DEFINITION.
PUBLIC SECTION.

TYPES ty_packed TYPE p LENGTH 4 DECIMALS 2.
METHODS set_height IMPORTING im_height TYPE ty_packed.
METHODS set_weight IMPORTING im_weight TYPE ty_packed.
METHODS set_bmi.
METHODS get_bmi RETURNING VALUE(r_bmi) TYPE ty_packed.
PRIVATE SECTION.
DATA : bmi TYPE I,
       height TYPE I,
       weight TYPE I.
ENDCLASS.
CLASS CL_PERSON IMPLEMENTATION.
METHOD set_height.
    height = im_height.
  ENDMETHOD.
  METHOD set_weight.
    weight = im_weight.
  ENDMETHOD.
  METHOD set_bmi.
   bmi = height / weight.
  ENDMETHOD.
  METHOD get_bmi.
    r_bmi = bmi.
  ENDMETHOD.
ENDCLASS.
```



```
DATA john TYPE REF TO cl_person.
DATA mark TYPE REF TO cl_person.
START-OF-SELECTION.
CREATE OBJECT john.

john->set_height( exporting im_height = 165 ).
john->set_weight( exporting im_weight = 50 ).
john->set_bmi( ).
WRITE : / 'John's BMI is', john->get_bmi( ).

CREATE OBJECT mark.

mark->set_height( exporting im_height = 175 ).
mark->set_weight( exporting im_weight = 80 ).
mark->set_bmi( ).
WRITE : / 'Mark's BMI is', mark->get_bmi( ).

IF john->get_bmi( ) GT mark->get_bmi( ).
WRITE : / 'John is fatter than Mark'.
ELSE.
WRITE : / 'Mark is fatter than John'.
ENDIF.
```

Listing 8.3 Using Objects

A class consists of two sections: a definition section and an implementation section. In the *definition section*, we define the *components* of the class, such as attributes, methods, and events. The *attributes* of a class store the state of an object, and the *methods* implement the logic to manipulate the behavior of an object by accessing the attributes.

Different *events* can be defined for the class, which can be raised during the runtime of the object. Methods can be called directly or can respond to events defined in the class. In the *implementation section*, we implement the methods to manipulate the behavior of an object or to respond to the events of the class.

Each class component can belong to a specific visibility section, such as *public*, *protected*, or *private*. The visibility section enables data encapsulation by putting restrictions on how the class component can be accessed. For example, a component belonging to the public visibility section can be accessed by external objects and programs without any restrictions, whereas a component belonging to the private visibility section can be accessed only by the methods of the same class.

Components belonging to the protected section can be accessed by the methods of the same class and its subclasses. A *subclass* inherits its components from a parent class. Once a class inherits from a parent class, it can access the public and protected components of the parent class. We'll learn more about encapsulation in Section 8.2.

In Listing 8.3, we defined the `CL_PERSON` class consisting of `HEIGHT`, `WEIGHT` and `BMI` attributes in the private visibility section. Because these attributes are private, they can't be accessed by external objects or programs directly. To manipulate these attributes, we defined setter and getter methods in the public section of the class. *Setter methods* allow you to set the value of an attribute, and *getter methods* allow you to read the value of an attribute. Defining the attributes in the private section and providing setter and getter methods to manipulate the attributes allows for more control, such as validating the data before changing the state of the object.

For example, we can avoid an invalid value being set for a person's height by validating the input in the `SET_HEIGHT` method. In the program, we defined two *object reference variables* called `JOHN` and `MARK` referring to the `CL_PERSON` class. An object exists as an instance of a class, and each class can have multiple instances. The object reference variable initially contains an empty value until the object is instantiated using the `CREATE OBJECT` statement. Once the object is instantiated, the object reference variable contains the reference (pointer) to the object in memory, and the object is accessed using the object reference variable. The object reference variables are also called *reference objects* and are defined using the `TYPE REF TO` addition.

Unlike in function modules, here each reference object occupies its own memory space and multiple instances of the class can be defined in the program, with each instance working with its own data.

In this section, we'll discuss local and global classes, class parts and components, and the use of visibility in classes.

Local and Global Classes

As previously discussed, classes can be defined locally in an ABAP program or globally in the system using Class Builder (Transaction SE24). *Local classes* can be used only within the program in which they're defined, whereas *global classes* can

be used in any ABAP program. Irrespective of whether the class is defined locally or globally, the syntax to define the class remains the same, as shown in Listing 8.3. Apart from visibility, there is no difference between a local class and a global class. When a class is used in an ABAP program, the system first searches if the class exists locally. If the class isn't defined locally, then the system looks for a global class.

When defining global classes, we can use the form-based editor of the Class Builder tool to easily define the components of the class and implement the methods without getting into the nitty gritty of the syntax.

Wherever applicable, we will show you the steps to define classes both locally and globally throughout this chapter. Define local classes if the scope of the object is limited to the program and the object may be irrelevant to other programs, but if you plan to use an object in multiple programs, we recommend defining the class globally in Class Builder.

Class Parts and Components

To break down the code in Listing 8.3 further, a class in ABAP has two sections:

- **Definition section**
  In the definition section, various components of a class, such as attributes, events, and methods, are defined. For example, in Listing 8.3, we defined the attributes and the methods of the class in the definition section of the class. When defining the methods, we also defined the parameter interface for the methods.

  For the sake of simplicity, we haven't defined any events in Listing 8.3, but if you want to trigger various events to which the methods of the class can respond, you can do so in the definition part of the class. The methods that respond to events are called *event handler methods*. We'll learn more about defining events and implementing suitable event handler methods in Section 8.1.4.
- **Implementation section**
  The implementation section of the class is where we implement the methods defined in the definition section. The methods are implemented between the `METHOD` and `ENDMETHOD` statements, and each method defined in the definition part must have an implementation in the implementation part of the class.

There are three important types of components available in a class:

- **Methods**
  Methods are used to change the state of an object and manipulate its behavior. For example, in Listing 8.3, different methods were implemented in the `CL_PERSON` class to set the height, weight, and BMI of the person. Methods can also be defined as event handler methods that respond to specific events during the runtime of the object. By using setter and getter methods, we can ensure that all attributes of the class are always accessed through its methods, thus guaranteeing data consistency.
- **Attributes**
  Attributes are the internal data fields of a class and store the state of an object. We can change the state of an object by changing the contents of an attribute.
- **Events**
  Events are defined in the definition area of a class and can be raised during the runtime of the object. A suitable event handler method is maintained in the implementation part of the class and is triggered when the corresponding event is raised. Events allow you to call specific methods dynamically during the runtime of the object.

  For example, if you want to perform specific tasks when the user double-clicks an output line, you can maintain a `DOUBLE_CLICK` event, which can be raised on that event. The registered event handler method will be called dynamically when the event is raised.

The components of a class can be categorized as follows:

- **Static components**
  Static components can be accessed directly using the class component selector `=>`. Static components behave similarly to function modules in that only one instance of a static component exists in the program.
- **Instance components**
  Instance components can't be accessed directly using the class name. To access an instance component, you need to define a reference object in the program using `TYPE REF TO` (for example, the objects `John` and `Mark` in Listing 8.3) and then access its attributes and methods through the reference object using the object component selector `->`.

  Before the attributes and methods can be accessed through the reference object, the reference object needs to be instantiated using the `CREATE OBJECT` statement, which creates an instance of the class (an object described by the class) in the program.

We'll discuss static and instance components further in Section 8.1.3.

Visibility

The components of a class can have different visibility sections, such as public, protected, and private. The visibility is set in the definition part of the class. Component visibility allows for data encapsulation and hiding the implementation from the external world.

Encapsulation is an important concept in OOP, and we'll discuss visibility further in Section 8.2.

8.1.2 Methods

A *method* can access other components of the same object. To access the attributes and methods of the same class, a special self-reference variable, `me`, is used. Even though the components of the class can be accessed directly from within the methods of the same class, for better readability it's recommended to use the self-reference variable `me` when accessing the same class components. You can access both static and instance components from within an instance method, whereas you only can access static components from within a static method.

To call a method of the class, we can either use the `CALL METHOD` statement (e.g., `CALL METHOD John->get_bmi`) or use the object component selector and parentheses directly (e.g., `john->get_bmi( )`).

A method can have importing, exporting, changing, and returning parameters. The importing, exporting, and changing parameters behave similarly to the parameters we define in function modules. For example, data is imported through importing parameters and exported through exporting parameters, while changing parameters can be used for both importing and exporting.

The methods that use returning parameters are called *functional methods*. Prior to SAP NetWeaver 7.4, functional methods could only have one returning parameter and no exporting parameters. However, with SAP NetWeaver 7.4, this restriction

is lifted. Functional methods are typically used when a result is returned to the caller program that can be directly assigned to a data object in the program.

For example, with the statement `v_bmi = oref->get_bmi( )`, `v_bmi` is the data object, `oref` is the reference object, and `get_bmi` is a method with a returning parameter the value of which will be assigned to the `v_bmi` variable.

Inheritance

Often, many objects are different, but similar enough to be put into a single class. For example, animals and humans have heartbeats, so a class may have a `get_heart_beat` method. However, dogs have tail lengths, so should this class also have a `get_tail_length` method? No, we can define another class as a child class (called a *subclass*) of the parent class. This subclass will have all the features of the parent class plus some additional features. This allows you to extend the functionality of your applications while leveraging the existing code.

Let's look at an example. Objects are designed to closely represent real-life objects—such as a printer. A printer may have an external interface, including a paper tray, keypad, and so on, and the basic functionality of the printer is to create a printout. OOP objects can be designed to represent that setup.

Therefore, let's create a class called `printer` with a method called `print`, as shown in Listing 8.4.

```
CLASS PRINTER DEFINITION.
PUBLIC SECTION.
METHODS print.
ENDCLASS.
CLASS PRINTER IMPLEMENTATION.
METHOD print.
WRITE 'document printed'.
ENDMETHOD.
ENDCLASS.
```

Listing 8.4 Parent Class for Printer

This is a basic printer, but what if we want to extend its functionality? For example, if we have a new printer model that has a counter feature to keep track of the number of copies printed, do we have to build all of the printer logic from scratch? No, we can define a subclass, as shown in Listing 8.5, that inherits all the functionality of the basic printer and extends it.


```
CLASS PRINTER_WITH_COUNTER DEFINITION INHERITING FROM PRINTER.
PUBLIC SECTION.
DATA counter TYPE I.
METHODS print REDEFINITION.
ENDCLASS.
CLASS PRINTER_WITH_COUNTER IMPLEMENTATION.
METHOD print.
Counter = counter + 1.
CALL METHOD SUPER->PRINT.
ENDMETHOD.
ENDCLASS.
```

Listing 8.5 Subclass Implementation

In Listing 8.5, the class `PRINTER_WITH_COUNTER` is inherited from the class `printer`. Here, `printer` is the superclass (parent class) and `PRINTER_WITH_COUNTER` is the subclass (child class). This process of a subclass inheriting the traits of a superclass is called *inheritance* in OOP.

Notice the keyword `INHERITING FROM` used in the definition of the class `PRINTER_WITH_COUNTER`. The subclass inherits all the components of the parent class. It will, by default, have all the attributes and methods defined in the parent class. If required, the methods inherited in the subclass can be redefined to add functionality.

In Listing 8.5, the `print` method in the `PRINTER_WITH_COUNTER` subclass is redefined to increment the counter attribute and calls the `print` method of the parent class. This effectively extends the functionality of the method while leveraging the existing functionality provided by the method of the parent class. Because the same `print` method behaves differently in the superclass and subclass, it leads to some interesting behavior called *polymorphism* (meaning *having many forms*), which we'll discuss in Section 8.4.

Now we have a printer with a counter, and we know that the counter increments by one every time a document is printed. However, what will its initial value be? How do we ensure that the counter always starts from one or any other initial value? These details are sorted out by defining a constructor for the class.

There are two types of constructors:

- **Instance constructor**
  An *instance constructor* is a special method called when the object is *instantiated*. The instance constructor is called for each object when it is instantiated.

- **Class constructor**
  The *class constructor* is special method called just once for the whole class in one internal mode. A class constructor can have a maximum of one instance and one static constructor. A static constructor can't have any parameters, whereas the instance constructor can have only importing parameters.

By defining importing parameters for the instance constructor, we can ensure that the object is always initialized before any of its attributes or methods are accessed.

Listing 8.6 shows the implementation of the constructor.

```
CLASS PRINTER_WITH_COUNTER DEFINITION INHERITING FROM PRINTER.
PUBLIC SECTION.
DATA counter TYPE I.
METHODS constructor IMPORTING count TYPE I.
METHODS print REDEFINITION.
ENDCLASS.

CLASS PRINTER_WITH_COUNTER IMPLEMENTATION.
METHOD constructor.
CALL METHOD super->constructor.
counter = count.
ENDMETHOD.
METHOD print.
counter = counter + 1.
CALL METHOD SUPER->PRINT.
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
DATA oref TYPE REF TO PRINTER_WITH_COUNTER.
CREATE OBJECT oref
EXPORTING
count = 0.
```

Listing 8.6 Class with Constructor

In Listing 8.6, the constructor method is defined in the definition part of the `PRINTER_WITH_COUNTER` class with one importing parameter, `count`. Because the `PRINTER_WITH_COUNTER` class is a subclass of the `printer` class, the superclass constructor must be called first in the implementation of the constructor method, even if there's no constructor defined in the superclass. This ensures that all of the superclass components in the inheritance tree are initialized properly. If a constructor is defined in the superclass, it won't be inherited by the subclass. This

allows each class in the inheritance tree to have its own constructor in order to initialize its attributes per requirements.

In the example in Listing 8.6, the `count` parameter of the constructor is assigned a value when instantiating the reference object `oref`. The syntax checker will give an error if the constructor's mandatory parameters aren't passed while instantiating the reference object. This ensures that the object is always instantiated properly.

Now, say we have a more advanced printer model that can print multiple copies along with the basic features provided so far. To create this printer, we'll inherit the `PRINTER_WITH_COUNTER` class and add additional functionality, as shown in Listing 8.7.

```
CLASS MULTI_COPY_PRINTER DEFINITION INHERITING FROM PRINTER_WITH_
COUNTER.
PUBLIC SECTION.
DATA copies TYPE I.
METHODS set_copies IMPORTING copies TYPE I.
METHODS print REDEFINITION.
ENDCLASS.

CLASS MULTI_COPY_PRINTER IMPLEMENTATION.
METHOD set_copies.
Me->copies = copies.
ENDMETHOD.
METHOD print.
Do copies TIMES.
CALL METHOD SUPER->Print.
ENDDO.
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
DATA oref TYPE REF TO MULTI_COPY_PRINTER.
CREATE OBJECT oref
EXPORTING
count = 0.
oref->set_copies( 5 ).
oref->print( ).
```

Listing 8.7 Class Extended to Print Multiple Copies

In Listing 8.7, we defined the `MULTI_COPY_PRINTER` class, inheriting from `PRINTER_WITH_COUNTER`. In the `MULTI_COPY_PRINTER` class, we defined a method called

`set_copies` that receives the `copies` importing parameter and assigns the value to the `copies` attribute. We redefined the `print` method to print multiple copies by calling the method in the superclass in a `DO` loop.

It's good practice not to allow direct access to the class attributes and to use methods to set or get attribute values. This gives us more control while manipulating attributes. The methods that set values are called *setter methods* and the methods that are used to get values are called *getter methods*. Setter methods are prefixed with `set_` and getter methods are prefixed with `get_`.

Encapsulation

Now, say our printer is an industrial printer and we want to set a restriction for the minimum number of prints. For example, the printer should only take a request if a minimum of fifty copies is requested. We implement this restriction by writing an `IF` condition in the `set_copies` method, as shown in Listing 8.8.

```
METHOD set_copies.
IF copies GE 50.
Me->copies = copies.
ELSE.
*Raise exception
ENDIF.
ENDMETHOD.
```

Listing 8.8 Restricting Copies in set_copies Method

The code in Listing 8.8 adds the restriction to set the `copies` attribute only if the value is greater than or equal to 50. However, there's one loophole in this implementation: Because the `copies` attribute is defined with its visibility as public, anyone can directly access this attribute from outside the class and set a value, thus bypassing the check in the `set_copies` method, as shown in Listing 8.9.

```
DATA oref TYPE REF TO MULTI_COPY_PRINTER.
CREATE OBJECT oref
EXPORTING
count = 0.
oref->copies = 5.
oref->print( ).
```

Listing 8.9 Bypassing set_copies Method to Set Number of Copies

As you can see in Listing 8.9, we can directly access the `copies` attribute to set the number of copies. This is where *encapsulation* and *implementation hiding* plays an

important role. By defining clear boundaries on what can be accessed from outside the class and what should be restricted, we not only make it easy for developers to use our classes, because they only need to know the public interface (components that are publicly accessible) of our class, but also ensure that inadvertent bugs don't creep into our software implementation.

Encapsulation also gives you the freedom to change the private sections of the class according to changing requirements without having to worry about dependencies. For example, if you change the interface of a method in the public visibility section, all the programs that call this method will be affected; however, because no programs can directly access the private section of a class, you can change private sections freely without having to worry about program dependencies.

In this example, we can make the attribute `copies` private so that it can be accessed only from the methods of the same class, as shown in Listing 8.10.

```
CLASS MULTI_COPY_PRINTER DEFINITION INHERITING FROM PRINTER_WITH_
COUNTER.
PUBLIC SECTION.
METHODS set_copies IMPORTING copies TYPE I.
METHODS print REDEFINITION.
PRIVATE SECTION.
DATA copies TYPE I.
ENDCLASS.
```

Listing 8.10 Making Attribute Private

As you can see, we made our printer more and more complex without having to duplicate any code. Also, as the printer added advanced capabilities, it still performed the basic task of printing a document. Therefore, a user using the advanced model of the printer need not be aware of its advanced features to perform the basic task of printing a document. This is similar to a car with advanced features like cruise control. The driver need not know how to set cruise control to drive the car, but he can learn to set cruise control to take advantage of it.

In addition, if the `print` method in the `printer` superclass is enhanced, then all the enhancements will be available by default to all the subclasses without having to change any of them.

8.1.3 Instance and Static Components

A class can contain both instance and static components. An instance attribute can have multiple instances in the program, whereas a static attribute has only one instance. An instance component is accessed through a reference object using the object component selector `->`, whereas a static component is accessed using the class component selector `=>`. Static components are preceded by the `CLASS` keyword during the definition.

Listing 8.11 shows the definition of both instance and static components. For simplicity, we're accessing the attributes of the class directly in this example, but in real-world code direct access to attributes should always be avoided by using setter and getter methods.

```
CLASS CL_PERSON DEFINITION.
  PUBLIC SECTION.
  CLASS-DATA height TYPE I.              "Static attribute
  DATA weight TYPE I.                    "Instance attribute
  METHODS get_bmi EXPORTING bmi TYPE I. "Instance method
  METHODS constructor                    "Instance constructor
      IMPORTING name TYPE CHAR20.
  CLASS-METHODS set_height         "Static Method
   IMPORTING height TYPE I.
  CLASS-METHODS class_constructor.      "Static Constructor
ENDCLASS.

CLASS CL_PERSON IMPLEMENTATION.
  METHOD class_constructor.
  ENDMETHOD.
  METHOD constructor.
  ENDMETHOD.
  METHOD get_bmi.
  ENDMETHOD.
  METHOD set_height.
  ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO cl_person. "Defining reference object
DATA v_bmi TYPE I.
START-OF-SELECTION.
*Accessing static attributes and methods using the selector "=>"
*with class reference
cl_person=>height = 165.
CALL METHOD cl_person=>set_height
EXPORTING
  height = 165.
*Instantiating the reference object and accessing the instance
```



```
*attributes and methods using the selector "->" with object
*reference.
CREATE OBJECT oref
EXPORTING
  name = 'JOHN'.
oref->weight = 50.
CALL METHOD oref->get_bmi
IMPORTING
  bmi = v_bmi.
```

Listing 8.11 Instance and Static Class Components

As shown, static components are defined with the preceding keyword `CLASS` (e.g., `CLASS-DATA`, `CLASS-METHODS`) and accessed using the `=>` selector with the class name reference directly (e.g., `class_name=>component`). Static methods are somewhat similar to function modules in that only one instance exists in the program memory and each subsequent call to the method points to the same instance, whereas each reference object works with its own instance components.

A static constructor can't have any parameter interface, whereas an instance constructor can have importing parameters. A static constructor is executed only once on the first access of any of the class components, including instantiation of any reference object. An instance constructor is executed each time a reference object is instantiated.

Table 8.1 compares and contrasts instance and static components.

Instance Component	Static Component
Instance components are accessed through a reference object with the `->` selector.	Static components can be accessed directly through the class name reference with the `=>` selector even before a reference object is instantiated. They can also be accessed using a reference object.
Multiple instances can exist.	Only single instance exists.
Within an instance method, both static and instance attributes can be accessed.	Within a static method, only static attributes or static methods can be accessed.
To access the instance components of a class, a reference object needs to be defined in the program and instantiated using `CREATE OBJECT` statement.	Static components can be accessed directly from program using a class reference (similar to calling a function module directly) without the need for a reference object.

Table 8.1 Differences between Instance and Static Components

Instance Component	Static Component
An instance constructor only can have importing parameters and is executed each time a reference object is instantiated.	A static constructor can't have any parameter interface and is executed only once per program session. However, it's executed before the instance constructor, if the class has both static and instance constructors, and when the first reference object is instantiated, provided it isn't loaded previously.

Table 8.1 Differences between Instance and Static Components (Cont.)

8.1.4 Events

Events can be defined in a class to be triggered from a method (called *triggers*) when a specific event takes place during the runtime of the object; specific methods (called *event handlers*) react to these events. For example, we can define an event called `double_click` in the class to be triggered in one of the methods when the user double-clicks the report output. The `on_double_click` method can be defined as an event handler for this event so that it's executed when the `double_click` event is triggered from the trigger method.

To trigger an event, the class must have the events defined in the definition part of the code and triggered in one of its methods. *Instance events* are defined using the `EVENTS` keyword and *static events* are defined using the `CLASS-EVENTS` keyword. A static event can be triggered from both static and instance methods, whereas, an instance event can only be triggered from an instance method.

Events can have exporting parameters that are passed to the event handler method when the event is triggered. To trigger an event, the statement `RAISE EVENT event_name` is used in one of the methods of the class. The `EXPORTING` addition is used with the `RAISE EVENT` keyword to pass the event parameters to the event handler method.

Event handlers are special methods that are defined to react to the event. Event handler methods are defined in the definition part of the class as methods for specific events. The syntax to define an event handler method is `METHODS method_name FOR EVENT event_name OF class_name`, where `method_name` is the name of the event handler method, `event_name` is the name of the event, and `class_name` is the name of the class in which the event is defined. If the event has exporting

parameters defined, then the event handler must import all non-optional parameters. All instance events have an implicit `sender` parameter that can be imported in the event handler. The `sender` parameter contains a reference to the sender object in which the event is defined. To define a static method as an event handler, use the `CLASS-METHODS` keyword.

To allow an event handler method to react to an event, we must determine at runtime the trigger to which it is to react. We can do that by using the `SET HANDLER` statement. The `SET HANDLER` statement registers the event handler methods with the event triggers.

Instance events can be registered to a specific instance or to all instances, whereas static events are registered to the whole class. The syntax to use `SET HANDLER` statement for instance events is as follows:

```
SET HANDLER event_handler_method FOR oref.
```

or:

```
SET HANDLER event_handler_method FOR ALL INSTANCES.
```

The syntax to use the `SET HANDLER` statement for static events is as follows:

```
SET HANDLER event_handler_method.
```

Here, `event_handler_method` is an instance or static method that is defined as an event handler for an event. The event handler method will not be triggered if the handler is not registered using the `SET HANDLER` statement.

Listing 8.12 shows an example to define and register events in a class. This example defines the `cl_events_demo` class with two events, `double_click` and `right_click`. Here, `double_click` is defined as an instance event and `right_click` is defined as a static event. The `double_click` event also has a couple of mandatory exporting parameters (`column` and `row`) defined. We've also defined a `trigger_event` method to act as a trigger method. This is defined similarly to any regular method. The `on_double_click` and `on_right_click` methods are defined as event handler methods for the `double_click` and `right_click` events, respectively.

```
CLASS cl_events_demo DEFINITION.
  PUBLIC SECTION.
    EVENTS double_click
    EXPORTING
      VALUE(column) TYPE i
      VALUE(row) TYPE i .
    CLASS-EVENTS right_click .
```

```
    METHODS trigger_event.
    METHODS on_double_click FOR EVENT double_click OF cl_events_demo
      IMPORTING
          column
          row
          sender.
    METHODS on_right_click FOR EVENT right_click OF cl_events_demo.
ENDCLASS.
CLASS cl_events_demo IMPLEMENTATION.
  METHOD trigger_event.
    RAISE EVENT double_click
       EXPORTING
         column = 4
         row    = 5.
  RAISE EVENT right_click.
  ENDMETHOD.
  METHOD on_double_click.
  WRITE: / 'Double click event triggered at column', column,
           'and row', row.
  ENDMETHOD.
  METHOD on_right_click.
    WRITE: / 'Right click event triggered'.
  ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO cl_events_demo.
START-OF-SELECTION.
  CREATE OBJECT oref.
  SET HANDLER oref->on_double_click FOR oref. "Instance event
  SET HANDLER oref->on_right_click. "Handler for Static event
  CALL METHOD oref->trigger_event( ).
```

Listing 8.12 Defining Events

In the implementation part of the class, the `trigger_event` method uses the `RAISE EVENT` statement to trigger the `double_click` and `right_click` events. Because the `double_click` event contains defined exporting parameters, this method exports certain values to the event handler method. The `on_double_click` method imports the values that were exported by the trigger method. In the program code, the events are registered using the `SET HANDLER` statement, and the code in the event handler methods is executed when the `trigger_event` method is called.

In this section, we introduced the basics of OOP, including classes, methods, instance and static components, and event. Working with objects simplifies the implementation of functionality. OOP concepts provide many advantages compared to procedural programming techniques. Programs developed using OOP techniques are easy to maintain and, if designed well, should be easy to enhance.

Now that you've been introduced to the basics of OOP, in the next few sections we'll discuss some of the characteristics of this kind of programming, beginning with the process of encapsulation.

8.2 Encapsulation

In procedural programming, developers generally use a technique called *functional decomposition* for application development design. In functional decomposition, the series of steps that the application needs to perform are identified from the functional requirements, and procedures are developed for each step. In other words, complex business requirements are broken down into smaller, easy to understand steps. Once all the procedures are in place, they are composed into the main program, which calls the procedures in a fixed order and passes the relevant data to each procedure.

For example, if you want to develop software for an ATM machine, then the requirements can be broken down into smaller steps, such as reading a card, verifying its PIN, dispensing cash, and so on. Modules can then be developed separately for each of these steps, such as one module to read the card, another module to validate the PIN, another module to verify the account balance and dispense cash, and so on. These modules can then be combined to form the overall application. By breaking down the requirements into smaller steps that are easy to understand and maintain, this approach makes it easy to develop small- to medium-sized applications.

However, because procedures (function modules or subroutines) don't have their own data environment, they're dependent on the main program to supply the data each time a procedure is called, or they must work with the global data of the program. This makes the procedures tightly coupled with the main program. As the application branches out and becomes more and more complex, it may lead to maintenance headaches.

For example, if many procedures are manipulating the global data, we can't be sure if the application will behave in a predictable way if the sequence of procedure calls is changed for any future enhancements. By depending entirely on the main program to supply the required data each time the procedure is called, the interface will soon be cluttered, and it makes the procedure too restrictive to work in other environments. The developer of the procedure depends too much

on the external application, over which he has no control. It also becomes difficult to develop loosely coupled procedures that can work seamlessly for future enhancements. In addition, once the procedure is developed, the developer can't change the parameter interface for future enhancements without breaking existing applications that call the procedure.

In OOP, because objects have their own data environment, they can be made smart enough to make their own decisions without too much dependency on external applications.

Encapsulation in OOP allows you to define boundaries and hide implementation from the external world. The term *encapsulation* means that the attributes (data) and the methods (functions) that manipulate the data are enclosed in a capsule (object). This means that we can set a boundary between what can be accessed from within the object and from the outside world. This boundary helps to address many of the issues mentioned previously regarding procedures.

In this section, we'll discuss some of the characteristics of encapsulation in OOP. We'll look at component visibility, the different visibility sections, friendship between classes, and implementation hiding.

8.2.1 Component Visibility

As shown back in Listing 8.7, the component of an object can be accessed from an *external program* or from a *subclass* or from within the class. The access restriction on a component of a class is what defines its visibility. In other words, we can decide if the component of a class can be accessed only from within the class or its subclasses or from external programs.

Back in Listing 8.10, we defined sections (`PUBLIC SECTION`, `PRIVATE SECTION`) under which components are declared. These sections are called *visibility sections*. There are three visibility sections: *public*, *protected*, and *private*.

The visibility section identifies how the component of a class can be accessed. The following list defines the different visibility sections:

- **Public section**
  The components in the *public section* can be accessed from within the class and outside of the class. These components form the public interface of the class.

- **Protected section**
  The components in the *protected section* can be accessed only from within the class and its subclasses (child classes). These components can't be accessed from external programs.
- **Private section**
  The components in the *private section* can be accessed only from within the class and its friend classes (we'll discuss friend classes in Section 8.2.2).

Listing 8.13 shows an example of the visibility sections.

```
CLASS cl_encapsulation_demo DEFINITION.
  PUBLIC SECTION.
    METHODS print.
    METHODS set_copies IMPORTING copies TYPE i.
    METHODS get_counter EXPORTING counter TYPE i.
  PROTECTED SECTION.
    METHODS reset_counter.
  PRIVATE SECTION.
    DATA copies TYPE i.
    DATA counter TYPE i.
    METHODS reset_copies.
ENDCLASS.
CLASS cl_encapsulation_demo IMPLEMENTATION.
  METHOD print.
    "Business logic goes here
  ENDMETHOD.
  METHOD set_copies.
    "Business logic goes here
    me->copies = copies.
  ENDMETHOD.
  METHOD get_counter.
    "Business logic goes here
    counter = me->counter.
  ENDMETHOD.
  METHOD reset_counter.
    "Business logic goes here
    CLEAR counter.
  ENDMETHOD.
  METHOD reset_copies.
    "Business logic goes here
    CLEAR copies.
  ENDMETHOD.
ENDCLASS.
CLASS cl_encapsulation_sub_demo DEFINITION INHERITING FROM cl_
encapsulation_demo.
  PROTECTED SECTION.
    METHODS reset_counter REDEFINITION.
```

```
ENDCLASS.
CLASS cl_encapsulation_sub_demo IMPLEMENTATION.
  METHOD reset_counter.
    super->reset_counter( ).
*    super->reset_copies( ). "gives syntax error
  ENDMETHOD.
ENDCLASS.
```

Listing 8.13 Implementing Visibility Sections

The code in Listing 8.13 implements the `cl_encapsulation_demo` class, and the `cl_encapsulation_sub_demo` class is defined as a subclass of `cl_encapsulation_demo`. The code shows the syntax to implement visibility sections. Notice the order of implementation of the visibility sections: First, public components are defined, then protected components, and finally private components. This declaration order can't be changed.

In Listing 8.13, the `cl_encapsulation_sub_demo` subclass can access only the public and protected components of its superclass, `cl_encapsulation_demo`. If the `cl_encapsulation_sub_demo` class tries to access the private components of its superclass, the attempt will result in a syntax error. Note that when redefining a method of the superclass in a subclass, the visibility section of the method can't be changed. For example, if the method is defined in the protected section of the superclass, then the method should be redefined in the protected section of the subclass.

Figure 8.1 shows the visibility set in Class Builder for global classes.

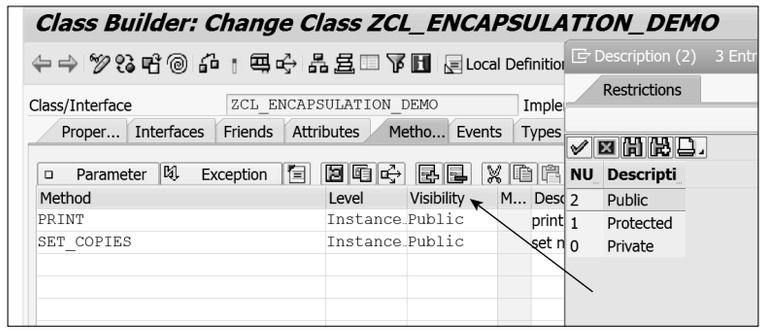


Figure 8.1 Visibility Using Class Builder

With visibility sections, clear boundaries can be defined so that only the required components of an object are exposed to the outside world. This arrangement

provides greater flexibility in designing the objects because the private and protected components of a class can be changed per changing business requirements without affecting the external programs, which only need to worry about the public interface.

By keeping object attributes private and implementing setter and getter methods in the class, we have more control and security over the object's data; any change to the attributes should be requested through the setter methods, and any request for information should be requested through getter methods. This arrangement allows for better abstraction and prevents undesired data corruption scenarios, which in turn helps objects perform in a predictable way.

For example, by implementing a setter method to set the value of an attribute, we can ensure that all the business logic is applied before the value is changed. This level of certainty isn't possible if the attribute is directly accessed by an external program to assign a value over which we have no control.

8.2.2 Friends

A class can declare another class as its friend to allow access to all of its components, including its private and protected components. A *friend class* is any class that is explicitly defined as a friend in the class definition. The friendship relation is not reciprocal; for example, if class `c1` declares class `c2` as a friend, then the methods of the class `c2` can access the private components of class `c1`, but the methods of class `c1` can't access the private components of class `c2`.

Listing 8.14 expands the code from Listing 8.13 to show the implementation of friend classes.

```
CLASS  c1 DEFINITION DEFERRED.
CLASS  c2 DEFINITION DEFERRED.
CLASS cl_encapsulation_demo DEFINITION FRIENDS c1 c2.
  PUBLIC SECTION.
    METHODS print.
    METHODS set_copies IMPORTING copies TYPE i.
    METHODS get_counter EXPORTING counter TYPE i.
  PROTECTED SECTION.
    METHODS reset_counter.
  PRIVATE SECTION.
    DATA copies TYPE i.
    DATA counter TYPE i.
    METHODS reset_copies.
ENDCLASS.
```

```
CLASS cl_encapsulation_demo IMPLEMENTATION.
  METHOD print.
    "Business logic goes here
  ENDMETHOD.
  METHOD set_copies.
    "Business logic goes here
    me->copies = copies.
  ENDMETHOD.
  METHOD get_counter.
    "Business logic goes here
    counter = me->counter.
  ENDMETHOD.
  METHOD reset_counter.
    "Business logic goes here
    CLEAR counter.
  ENDMETHOD.
  METHOD reset_copies.
    "Business logic goes here
    CLEAR copies.
  ENDMETHOD.
ENDCLASS.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    METHODS get_counter IMPORTING counter TYPE i.
  PROTECTED SECTION.
    DATA counter TYPE i.
ENDCLASS.
CLASS c1 IMPLEMENTATION.
  METHOD get_counter.
    DATA oref TYPE REF TO cl_encapsulation_demo.
    CREATE OBJECT oref.
    oref->reset_counter( ).
    oref->counter = counter.
  ENDMETHOD.
ENDCLASS.
CLASS c2 DEFINITION.
  PUBLIC SECTION.
    METHODS set_copies IMPORTING copies TYPE i.
  PRIVATE SECTION.
    DATA copies TYPE i.
ENDCLASS.
CLASS c2 IMPLEMENTATION.
  METHOD set_copies.
    DATA oref TYPE REF TO cl_encapsulation_demo.
    CREATE OBJECT oref.
    oref->reset_copies( ).
    oref->copies = copies.
  ENDMETHOD.
ENDCLASS.
```



```
CLASS cl_encapsulation_sub_demo DEFINITION INHERITING FROM cl_
encapsulation_demo.
  PROTECTED SECTION.
    METHODS reset_counter REDEFINITION.
ENDCLASS.
CLASS cl_encapsulation_sub_demo IMPLEMENTATION.
  METHOD reset_counter.
    super->reset_counter( ).
    super->reset_copies( ). "Gives syntax error
  ENDMETHOD.
ENDCLASS.
```

Listing 8.14 Visibility with Friends

The code in Listing 8.14 implements the `cl_encapsulation_demo` class and defines the `c1` and `c2` classes as friends via the `FRIENDS` addition in the `CLASS DEFINITION` statement. Any number of classes can be listed with the `FRIENDS` addition, separated by spaces. Also notice the `CLASS c1 DEFINITION DEFERRED` and `CLASS c2 DEFINITION DEFERRED` statements at the beginning of the code. These two statements tell the syntax checker that the classes `c1` and `c2` are defined later in the code; otherwise, the syntax checker will give an error when the `cl_encapsulation_demo` class refers to the `c1` and `c2` classes as friends in its definition.

In Listing 8.14, the `c1` and `c2` classes can access the public, private, and protected components of `cl_encapsulation_demo` because they're defined as friends for that class. However, the `cl_encapsulation_demo` class can't access the protected or private sections of the `c1` or `c2` classes because the relationship isn't reciprocal. Without this restriction, any class can declare itself as a friend of another class to access its private or protected components, which defeats the purpose of having visibility sections.

Figure 8.2 shows the friend classes defined under the FRIENDS tab in Class Builder, where you can maintain friend classes for a global class.

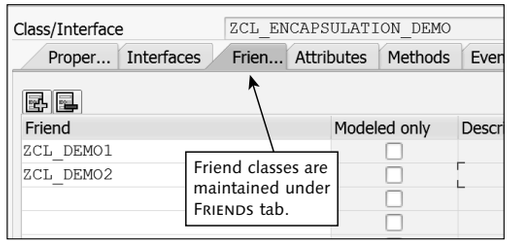


Figure 8.2 Friends Definition in Class Builder

8.2.3 Implementation Hiding

Hiding implementation from the outside world allows for better implementation control while providing flexibility for future enhancements due to ever-changing business requirements. Keeping the public interface to a minimum allows you to change the private and protected sections of the class per changing requirements without having to worry about existing applications that call the class. This saves a lot of headaches during enhancements and maintenance.

Let's consider a use case to understand how implementation hiding is useful. For this example, assume you're developing an app for a mobile operating system such as iOS or Android. One of the features in this software is to show notifications to the user from different apps. To facilitate this, define a public attribute in the application programming interface (API) class that can be accessed by app developers to send notifications. However, this way, you can't have control over the number of notifications displayed by the third-party app or control any spam activity. Listing 8.15 shows the sample API code.

```
CLASS cl_notification_api DEFINITION.
PUBLIC SECTION.
DATA message TYPE string.
METHODS set_message IMPORTING im_message TYPE string.
METHODS display_notification.
ENDCLASS.
CLASS cl_notification_api IMPLEMENTATION.
METHOD set_message.
message = im_message.
ENDMETHOD.
METHOD display_notification.
WRITE message.
ENDMETHOD.
ENDCLASS.
*Code in the calling program.
DATA notify TYPE REF TO cl_notification_api.
CREATE OBJECT notify.
notify->set_message( im_message = 'My App notification' ).
Notify->display_notification( ).
```

Listing 8.15 Making All Implementation Public

In Listing 8.15, the `cl_notification_api` class is defined with a `message` attribute and the `set_message` and `display_notification` methods. Let's assume `cl_notification_api` works as an API that can be called from third-party apps to display notifications to the user.

The attribute `message` holds the message that can be set by calling the `set_message` method. The notification is displayed by calling the `display_notification` method. In this class, all of the components have public visibility.

Because all of the components are visible outside the class, any change to them will break the implementation in all the apps that use this API. For example, implementing additional features such as filtering a message or restricting the number of messages per app may be difficult with this design.

The `set_message` setter method is provided to set the message after screening it (such as screening for inappropriate words). However, this restriction can be bypassed easily by third-party app developers, because the `message` attribute is also accessible from outside the class. Therefore, outside developers can directly access the attribute to set the message instead of calling the `set_message` method.

The API in Listing 8.15 can be enhanced by hiding the implementation, as shown in Listing 8.16.

```
CLASS cl_notification_api DEFINITION.
  PUBLIC SECTION.
  METHODS set_message IMPORTING im_message TYPE string.
  METHODS display_notification.
  PRIVATE SECTION.
  DATA MESSAGE TYPE string.
  METHODS filter_message RETURNING VALUE(boolean) TYPE boolean.
  METHODS check_count RETURNING VALUE(boolean) TYPE boolean.
  METHODS check_status RETURNING VALUE(boolean) TYPE boolean.
ENDCLASS.
CLASS cl_notification_api IMPLEMENTATION.
  METHOD set_message.
    MESSAGE = im_message.
  ENDMETHOD.
  METHOD display_notification.
    IF me->filter_message( ) EQ abap_true OR
       me->check_count( ) EQ abap_true OR
       me->check_status( ) EQ abap_true.
    WRITE MESSAGE.
    ELSE.
      CLEAR message.
    ENDIF.
  ENDMETHOD.
  METHOD filter_message.
*Filtering logic goes here and the parameter "Boolean" is set to
*abap_true or abap_false accordingly.
  ENDMETHOD.
  METHOD check_count.
```

```
*Logic to check number of messages goes here and the parameter
*"Boolean" is set to abap_true or abap_false accordingly.
  ENDMETHOD.
  METHOD check_status.
*Logic to check user personal setting goes here and the parameter
*"Boolean" is set to abap_true or abap_false accordingly.
  ENDMETHOD.
ENDCLASS.
*Code in the calling program.
DATA notify TYPE REF TO cl_notification_api.
CREATE OBJECT notify.
notify->set_message( im_message = 'My App notification' ).
Notify->display_notification( ).
```

Listing 8.16 Hiding Implementation

The code in Listing 8.16 now moves the `message` attribute to the private section so that it can't be accessed outside of the class, meaning that its implementation is hidden from the outside world. In addition, three new methods are defined in the private section to perform various checks on the message before it's displayed to the user; for example, if the user has turned off notifications for the app, the program can check for that information in the `check_status` method, and the message is not displayed if the `check_status` method returns the Boolean value `abap_false`. Similarly, if the user has set a restriction not to display more than five notifications per day for an app, the `check_count` method can handle that validation. The validating methods used in the private section can be enhanced or changed without having to worry about any dependencies.

External developers need not know about the intricacies of the validations performed to display their notifications. All they need to know is how to set the message and call the notification API. With the implementation in Listing 8.16, you have total control over the displayed notifications, and there's no way for any checks to be bypassed. This also makes the implementation future-proof, because you can add as many additional checks as you'd like without external developers changing anything about the way the API is called in their applications.

When used the right way, implementation hiding makes software development less prone to errors and easier to enhance or maintain.

Now that we've looked at the primary characteristics of encapsulation, in the next section we'll move onto another OOP concept: inheritance.

8.3 Inheritance

Inheritance is an important concept in OOP that helps us avoid repetition of code and leverage existing code libraries. In Section 8.1, we described how inheritance makes it easy to expand functionality without having to rewrite existing code. With procedural programming techniques, the closest you can get to something similar to inheritance is either editing the existing code to expand the functionality or copying the existing code and then adding the additional functionality. Both approaches are undesirable, because editing existing code may break the original functionality, and it requires a lot of testing to check that the original and the new functionality both work correctly, which can itself be a time-consuming process to go through each time the functionality is enhanced. If we copy the existing code, not only is the code now redundant, but also it leads to a maintenance nightmare, because each update to the original code needs to be replicated in all of the copies.

Inheritance allows you to expand a class to meet more specific requirements while reusing what's already been developed without having to repeat the existing code. For example, say that during the initial stage of development you defined a class to process information about students. Because your requirement didn't differentiate between types of students, you developed a generic student class.

However, later in the development process, the requirement has been enhanced to branch students into various categories—commerce students, IT students, science students, and so on—with each student category required to perform specific tasks. As the requirement becomes more specific, you can simply inherit the original student class to add more specific functionality for each student category, as shown in Listing 8.17. This allows you to move from generic functionality to specific functionality without breaking the existing implementation while reusing the effort that already went into the original development.

```
CLASS cl_student DEFINITION.
  PUBLIC SECTION.
  METHODS tuition_fee.
ENDCLASS.
CLASS cl_commerce_student DEFINITION INHERITING FROM cl_student.
PUBLIC SECTION.
METHODS tuition_fee REDEFINITION.
METHODS subject. "New method for additional functionality
ENDCLASS.
```

```
CLASS cl_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
  "Logic to calculate tuition fee for commerce students goes here
  ENDMETHOD.
METHOD subject.
ENDMETHOD.
ENDCLASS.
```

Listing 8.17 Inheritance

In Listing 8.17, the new `cl_commerce_student` class is a subclass of the original student superclass from which it's inheriting, `cl_student`.

In this section, we'll look at the different components of the inheritance concept. We'll then look at both abstract and final classes and methods before moving on to composition relationships. In the final subsection, we'll look at how to use the refactoring assistant in Class Builder to move class components in the inheritance tree.

8.3.1 Inheriting Components

In Listing 8.16, we were only concerned with what can be accessed from within and outside of a class. The public section allowed access from external applications, while the private section restricted access to components within the class.

Inheritance brings a new dimension to implementation hiding. With the protected section, you can restrict access to the components from external applications but allow access to the subclasses. Therefore, for external programs, the components defined in the protected section are similar to the components in the private section. This allows you to provide access to child classes without exposing the components of a parent class to the public.

Listing 8.18 provides some sample code to demonstrate inheritance. In this example, we're demonstrating how all the components of the parent class are inherited by the child class.

```
CLASS cl_parent DEFINITION.
PUBLIC SECTION.
METHODS set_value IMPORTING value TYPE string.
PROTECTED SECTION.
DATA value TYPE string.
METHODS check_value.
PRIVATE SECTION.
METHODS reset_value.
```



```
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
METHOD set_value.
ENDMETHOD.
METHOD check_value.
ENDMETHOD.
METHOD reset_value.
ENDMETHOD.
ENDCLASS.
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
ENDCLASS.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT child.
Child->set_value( value = 'child' ).
```

Listing 8.18 Inheritance Example

In Listing 8.18, the `cl_child` class is defined as a subclass of `cl_parent`. Here, `cl_parent` is the superclass, and it implements the `set_value` method in the public section, the `check_value` method and the `value` attribute in the protected section, and the `reset_value` method in the private section.

The `cl_child` class is inherited from `cl_parent` using the `inheriting from` addition of the `class definition` statement. This sets the parent-child relationship between the two classes. The `cl_child` class doesn't list any components on its own. All the components in the public and protected sections of the `cl_parent` superclass are, by default, available to the `cl_child` subclass. This is shown in the program code under `start-of-selection` (see Listing 8.18), where a reference object child is defined by referring to `cl_child` and the `set_value` method of `cl_parent` is accessed using this child class reference with the statement `Child->set_value( value = 'child' )`.

Keep the following important points about inheritance in mind:

- The subclass can access the components defined under public and protected visibility sections in the superclass.
- You can't define a component in a subclass with the same name as the component in the public or protected sections of the superclass.
- Because the private section of the superclass is invisible to the subclass, you can define a component in the subclass with the same name as the component in the superclass.

- The visible instance attributes in the superclass can be accessed from the subclass directly or by using the self-reference variable `me`.
- The visible instance methods of the superclass can be accessed from the subclass using the pseudo-reference variable `super`.
- The static attributes of a class are associated with the complete inheritance tree (all the classes in the hierarchy) and not just with a single class, so they can be accessed using any class reference in the hierarchy. For example, if three classes `A`, `B`, and `C` are in inheritance relationship (`B` inherits `A` and `C` inherits `B`) and class `A` contains a static attribute `counter`, then this attribute can be accessed as `A=>counter` or `B=>counter` or `C=>counter`.
- The constructor of the superclass is not inherited by the subclass. This allows the subclass to define its own constructor, but to ensure the components of the superclass are instantiated properly, it's mandatory to call the constructor of the superclass in the constructor of the subclass, for instance constructors. If the subclass implements a static constructor, the runtime environment automatically calls the static constructor of the superclass when the subclass is instantiated if the static constructor of the superclass isn't yet called.
- The constructor of the class will have visibility to its own components. For example, if a method is redefined in the subclass and this method is accessed in the constructor of both the superclass and subclass, then the constructor of the subclass will execute the redefined method in the subclass, while the constructor of the superclass will execute the original method in the superclass, as shown in Listing 8.19.
- A method of the superclass can be redefined in the subclass using the `REDEFINITION` addition to the `METHODS` statement in the definition part of the subclass, as shown in Listing 8.19. This allows you to enhance the functionality of the superclass method in the subclass.
- Any changes in the superclass will be automatically available to the subclass, whereas changes to subclasses don't affect the superclass. For example, if a method of the superclass is enhanced, then the enhancements will be automatically available to the subclass. However, if the same method is changed (redefined) in the subclass, then it won't impact the superclass.
- The definition of a component of superclass can't be changed in the subclass; in other words, you can redefine a superclass method in a subclass, but you can't change its parameter interface.


```
CLASS cl_parent DEFINITION.
  PUBLIC SECTION.
  METHODS constructor.
  PROTECTED SECTION.
  METHODS meth.
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
  METHOD constructor.
    Me->meth( ).
  ENDMETHOD.
  METHOD meth.
    WRITE 'I'm in parent class'.
  ENDMETHOD.
ENDCLASS.
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
  METHODS constructor.
  PROTECTED SECTION.
  METHODS meth REDEFINITION.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
  METHOD constructor.
    super->constructor( ).
    Me->meth( ).
  ENDMETHOD.
  METHOD meth.
    WRITE 'I'm in child class'.
  ENDMETHOD.
ENDCLASS.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT child.
```

Listing 8.19 Constructor Access in Superclass and Subclass

In Listing 8.19, `cl_parent` is defined with an instance constructor and the `meth` method. `cl_child` inherits the `cl_parent` class, and the `meth` method is redefined in the `cl_child` subclass. A reference object, `child`, is defined in the program referring to `cl_child`. When the reference object `child` is instantiated under `start-of-selection` using the `CREATE OBJECT` statement, it will call the constructor of the `cl_child` class. However, because it's mandatory to call the parent constructor before any instance components can be accessed, the constructor in `cl_child` calls the parent constructor using the `super->constructor( )` statement.

The `meth` method is called in the respective constructors of the superclass and subclass. The constructor of the `cl_child` subclass calls the redefined `meth`

method in `cl_child`, whereas the constructor of the `cl_parent` superclass calls the `meth` method defined in `cl_parent`. You can keep a breakpoint on the `CREATE OBJECT` statement and execute the code to see the sequence of the code execution in debug mode for easy understanding.

For global classes in Class Builder, the inheritance relation can be maintained either while creating the class, as shown in Figure 8.3, or under the PROPERTIES tab, as shown in Figure 8.4.

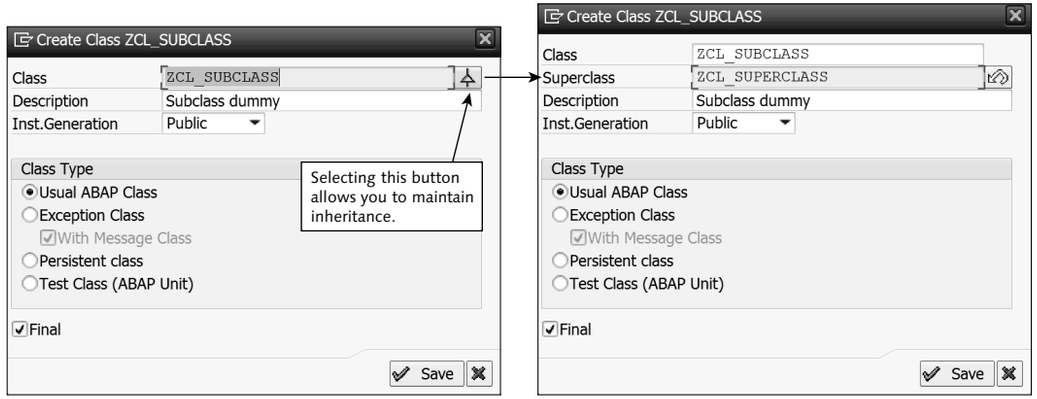


Figure 8.3 Maintaining Superclass Details in Class Builder

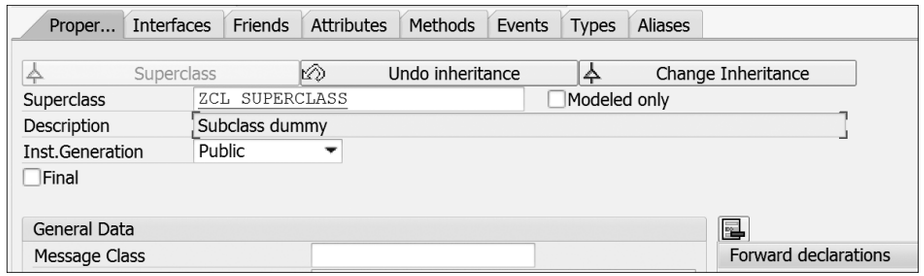


Figure 8.4 Maintaining Inheritance under Class Properties

To redefine a method in the subclass, you can use the REDEFINE button in Class Builder, as shown in Figure 8.5.

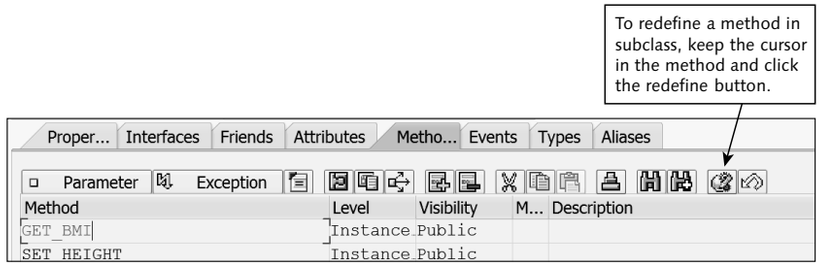


Figure 8.5 Redefine Button in Class Builder

8.3.2 Abstract Classes and Methods

Sometimes, you may want to define a generic class that can be used as a template that can be implemented in subclasses. Defining a dummy superclass with dummy methods may not be a good way of defining this template. For example, if you have students and various course categories and each student course category has a different process to determine its tuition, then you can define a student class with a `tuition_fee` method that can be used as a template, and its implementation can be maintained more specifically in the specific subclasses that inherit this class. For example, a `commerce_student` class can inherit the student class and implement the `tuition_fee` method specific to commerce students.

If you define this student class as a regular class, then you also need to maintain its implementation. However, because this is a generic class, maintaining the implementation doesn't make sense. In such scenarios, we can define this student class as an abstract class and define the `tuition_fee` method as an abstract method.

An *abstract class* only maintains a definition; no implementation is required for such a class if it contains all abstract components, which are inherited in a subclass in which the specific implementation can be maintained. Listing 8.20 provides an example of using an abstract class.

```
CLASS cl_student DEFINITION ABSTRACT.
  PUBLIC SECTION.
  METHODS tuition_fee ABSTRACT.
ENDCLASS.

CLASS cl_commerce_student DEFINITION INHERITING FROM cl_student.
PUBLIC SECTION.
```

```
METHODS tuition_fee REDEFINITION.
ENDCLASS.
CLASS cl_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
    "Logic to calculate tuition fee for commerce students goes here
  ENDMETHOD.
ENDCLASS.

CLASS cl_science_student DEFINITION INHERITING FROM cl_student.
  PUBLIC SECTION.
  METHODS tuition_fee REDEFINITION.
ENDCLASS.
CLASS cl_science_student IMPLEMENTATION.
  METHOD tuition_fee.
    "Logic to calculate tuition fee for science students goes here
  ENDMETHOD.
ENDCLASS.
```

Listing 8.20 Using Abstract Class

In Listing 8.20, `cl_student` is defined as an abstract class by using the `ABSTRACT` addition with the `CLASS DEFINITION` statement.

An abstract method is defined in class `cl_student` using the `ABSTRACT` addition with the `METHODS` statement. Because this class contains an abstract method, it can only be implemented in a subclass by using the `redefinition` addition. If the class contains a regular method (not an abstract method), then we need to maintain the implementation of that method in the implementation part of the class `cl_student`. This implementation then would be part of the template that the subclasses inherit. Because there are no regular methods in Listing 8.20, we've ignored the implementation of the `cl_student` class completely.

The code in Listing 8.20 defines two subclasses—`cl_commerce_student` and `cl_science_student`—which are inherited from the `cl_student` abstract class. The `tuition_fee` method is redefined in the respective subclasses to implement specific functionality.

Note

It isn't possible to create an instance of an abstract class, because an abstract class is only used as a template.

For global classes, the abstract property for a class is set under the PROPERTIES tab in the INST.GENERATION field, as shown in Figure 8.6.

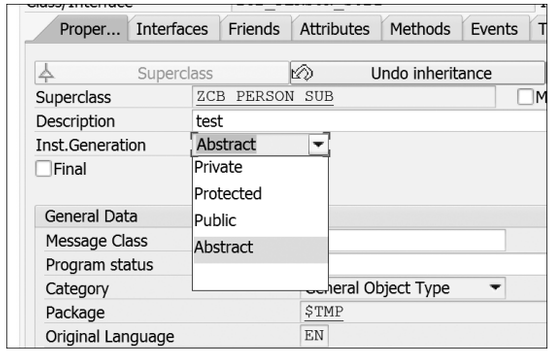


Figure 8.6 Abstract Class in Class Builder

For methods in Class Builder, the abstract property is set by keeping the cursor on the method, selecting the DETAIL VIEW button, and then selecting the ABSTRACT checkbox, as shown in Figure 8.7.

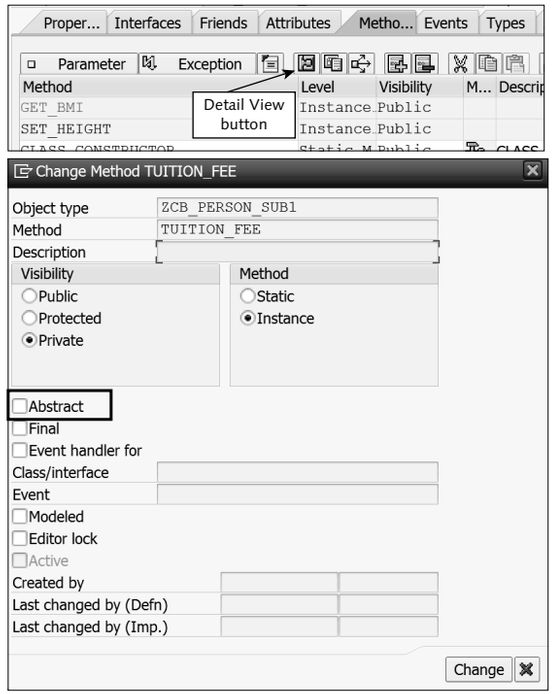


Figure 8.7 Defining Abstract Method in Class Builder

8.3.3 Final Classes and Methods

Sometimes, in the inheritance tree, it may not make sense to expand a class further. In such a scenario, you can make the class *final*, which means it can't be inherited further. A local class can be defined as a final class by adding `FINAL` to the `CLASS DEFINITION` statement, as shown in Listing 8.21.

```
CLASS cl_final DEFINITION FINAL.
.....
ENDCLASS.
```

Listing 8.21 Defining Final Class

For global classes, to define a class as final in Class Builder, the FINAL checkbox needs to be selected in the PROPERTIES tab, as shown in Figure 8.8.

Methods can also be defined as final so that they can't be redefined further in the subclass. For example, it may make sense to further inherit a class, but a method in the class may not need to be extended further because it's purpose is complete. You can make a method final by adding `FINAL` to the `METHODS` statement, as shown in Listing 8.22.

```
CLASS cl_parent DEFINITION.
PUBLIC SECTION.
METHODS student FINAL.
ENDCLASS.
```

Listing 8.22 Final Method

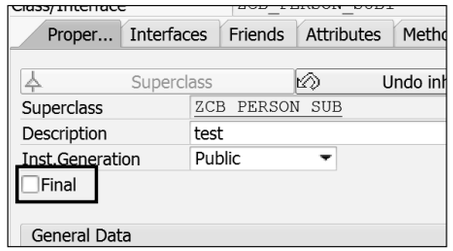


Figure 8.8 Final Class

To make the method of a global class final, keep the cursor on the method in the Class Builder and select the DETAIL VIEW button. In the CHANGE METHOD window, select the FINAL checkbox, as shown in Figure 8.9.

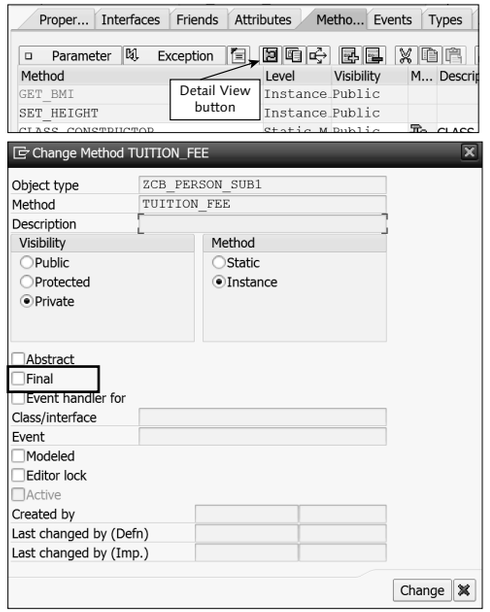


Figure 8.9 Final Method in Class Builder

8.3.4 Composition

Using inheritance, we design classes that fit an *is a* relationship. For example, a commerce student *is a* type of student, so we define the class `commerce_student` as a subclass of `student` in the example in Listing 8.20. Sometimes, in an effort to reuse the existing code, developers create an inheritance tree that doesn't fit an *is a* relationship. For example, if we have an existing `orders` class, it makes sense to create a `sales_order` class inheriting from the `orders` class, because a sales order *is an* order.

However, if you want to define a class for `delivery`, it may not make sense to inherit the `orders` class, because a delivery *is not* an order. However, each order *has* one or more deliveries associated with it. If the objects fit the *has a* relationship, we call it a *composition relationship*.

Composition allows you to reuse existing functionality by maintaining the existing object as an attribute in the class. Therefore, the `orders` class can have `delivery` as an attribute in the class. This arrangement allows you to take advantage of the existing functionality in the system, as shown in Listing 8.23.

```
CLASS cl_delivery DEFINITION.
PUBLIC SECTION.
METHODS get_delivery.
ENDCLASS.
CLASS cl_delivery IMPLEMENTATION.
METHOD get_delivery.
ENDMETHOD.
ENDCLASS.
CLASS cl_orders DEFINITION.
PUBLIC SECTION.
METHODS track_order.
PRIVATE SECTION.
DATA delivery TYPE REF TO cl_delivery.
ENDCLASS.
CLASS cl_orders IMPLEMENTATION.
METHOD track_order.
CREATE OBJECT delivery.
delivery->get_delivery( ).
ENDMETHOD.
ENDCLASS.
```

Listing 8.23 Composition Example

In Listing 8.23, the `cl_delivery` class is maintained as an attribute in the `cl_orders` class. The `delivery` object is instantiated in the `track` method to access the delivery information about the order.

Tip

As a rule of thumb, use an inheritance relationship between objects if they fit the *is a* relationship and use composition if the objects fit the *has a* relationship. This tip should help you make better design decisions.

8.3.5 Refactoring Assistant

When you're designing an application, sometimes you may miss defining the class components at the right level in the inheritance hierarchy. For example, you may have defined a method in the subclass that may make more sense in the superclass due to a change in the requirement.

In such a scenario, you can use the refactoring assistant tool in Class Builder to move the class components up or down the inheritance tree. This saves a lot of effort and any chance of missing steps when manually moving the components.

To use the refactoring assistant, select the menu path Utilities • Refactoring • Refactoring in Class Builder, as shown in Figure 8.10.

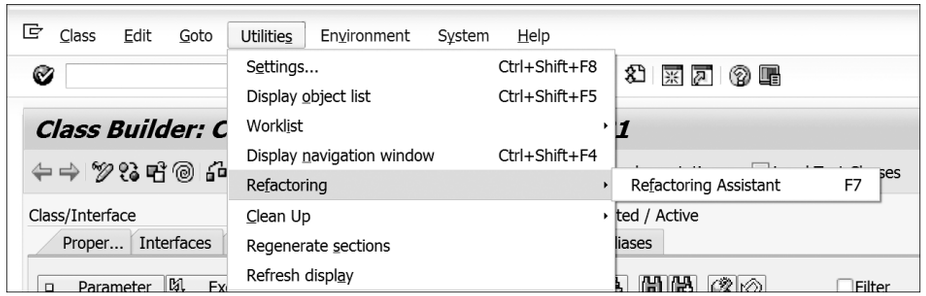


Figure 8.10 Refactoring Assistant Menu Path

In the Refactoring Assistant window (see Figure 8.11), you can drag and drop components to the right hierarchy level.

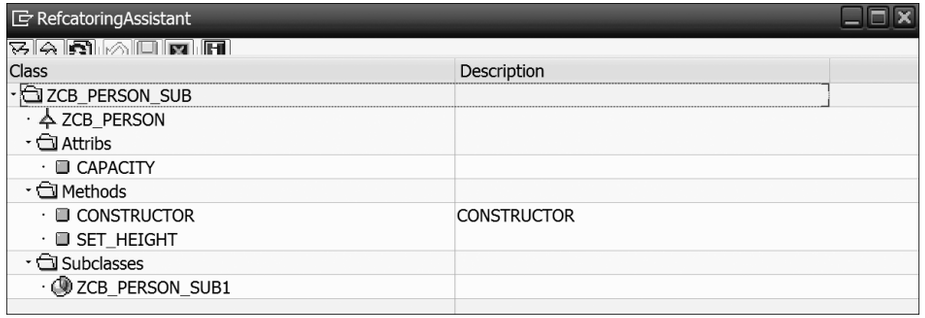


Figure 8.11 Refactoring Assistant Window

Now that we've walked through the inheritance principles, let's move on to using polymorphism in OOP.

8.4 Polymorphism

Polymorphism means *having many forms.* The inheritance concept leads to many interesting scenarios in which objects can take different forms. For example, it's possible for a subclass to respond to a method call of the superclass.

To understand polymorphism, you first need to understand static and dynamic types. We'll then look at how to use casting when a static type source reference object isn't the same as the target reference object. From there, we'll discuss how to use dynamic binding with the `CALL` method before finally looking at implementing multiple inheritances using interfaces.

8.4.1 Static and Dynamic Types

The *static type* of an object (reference variable) is the class type that is used to define an object. For example, in Listing 8.24, the `oref` reference is defined by referring to the `cl_class` class. In this case, the static type of the `oref` object is the `cl_class` class as it's statically defined in the code.

```
CLASS cl_class DEFINITION.
PUBLIC SECTION.
METHODS meth.
ENDCLASS.
CLASS cl_class IMPLEMENTATION.
METHOD meth.
ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO cl_class.
```

Listing 8.24 Code Defining Reference Object

Sometimes, you may want to assign one reference object to another reference object that doesn't share the same static type. For example, because instances of classes in an inheritance tree are interchangeable, you can assign the instance of the subclass to the instance of the superclass, as shown in Listing 8.25, by using the `parent = child` statement.

```
CLASS cl_parent DEFINITION.
  PUBLIC SECTION.
  METHODS meth1.
  METHODS meth2.
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
  METHOD meth1.
   WRITE 'In method1 of parent'.
  ENDMETHOD.
  METHOD meth2.
   WRITE 'In method2 of parent'.
  ENDMETHOD.
ENDCLASS.
```



```
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO cl_parent.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child.
Parent->meth2( ).
parent = child.
parent->meth2( ).
```

Listing 8.25 Example of Casting

In Listing 8.25, once the reference object `parent` is instantiated using the `CREATE OBJECT` statement, it sets the pointer (in memory) of the `parent` reference object to the `cl_parent` class. The pointer to which the reference object points at runtime is called the *dynamic type* of the object.

At this point, both the static type and dynamic type of the `parent` object are the same. The `parent = child` statement assigns the instance of the `child` object to the `parent` object. When one reference object is assigned to another reference object, the object itself is not assigned; only the pointer of the object is moved. In other words, after the assignment `parent = child`, the `parent` object points to the memory location of the `child` object. After this assignment, the dynamic type of the `parent` object is changed to the `cl_child` class and the static type is `cl_parent`.

At this point in the code, both the `parent` and `child` reference objects are pointing to the same `cl_child` object. Because there's no reference to the `parent` object after this assignment, the `parent` object will be removed from the memory automatically by the garbage collector. The ABAP runtime environment calls the *garbage collector* periodically to remove objects from memory that are no longer referenced by object reference variables.

In Listing 8.25, when the `parent->meth2( )` method is first called before the `parent = child` assignment (that is, before its dynamic type is changed), it will execute the code in the `meth2` method of the `cl_parent` superclass. However, the same statement after the `parent = child` assignment will execute the code in the `meth2` method of the `cl_child` subclass, effectively displaying polymorphism because the same object behaves differently during the runtime of the program.

8.4.2 Casting

Similar to the conversion rules applied when different types of data objects are assigned to each other, the assignment of objects of different static types are governed by certain rules. If the static type of the source reference object is not the same as the target reference object, then a special operation called a *cast* must occur. This process is known as *casting*. For the cast operation to work, the static type of the target reference object should be same or more general than the dynamic type of the source reference object.

In the example in Listing 8.25, the `cl_parent` parent object is more generic than the `cl_child` child object (a superclass will always be the same as or more generic than a subclass). In other words, the parent class has implemented two methods, whereas the subclass has become more specific by implementing a third method not available in the parent class. Therefore, the parent class is more generic than the child class, and for this reason, we could assign the child object to the parent object.

There are two different types of cast operations possible: a *narrowing cast* (or *up cast*) and a *widening cast* (or *down cast*). The following subsections look at these different casting operations.

Narrowing Cast

A narrowing cast occurs when the static type of the target reference object is more generic than the static type of the source reference object. The casting operation we performed in Listing 8.25 is a *narrowing cast*. It's called a narrowing cast because the parent class is more generic, and after casting the parent class can only access the components of the child class that are defined in the static type of the parent class, effectively reducing the scope of (narrowing) the components that can be accessed.

For example, say a parent class has two methods, `meth1` and `meth2`, and the child class has an additional method, `meth3`, which isn't defined in the parent class. After the casting operation, you can only access the `meth1` and `meth2` methods of the child class by using the parent object reference. If you try to access the `meth3` method, it will result in a syntax error, as shown in Listing 8.26. You can, however, access the `meth3` method using the child object reference, because it still exists in the program and isn't truncated.

```
Parent = child. "Assigns child reference to parent.
Parent->meth1( )."Calls the method meth1 in child if redefined.
Parent->meth2( )."Calls the method meth2 in child if redefined.
Parent->meth3( )."Results in syntax error.
Child->meth3( )."Works as expected.
```

Listing 8.26 Narrow Cast

The narrow cast can also be performed during instantiation by adding `TYPE` to the `CREATE OBJECT` statement as shown:

```
DATA parent TYPE REF TO cl_parent.
CREATE OBJECT parent TYPE cl_child.
```

Widening Cast

A *widening cast* occurs when the static type of the target reference object is more specific than the static type of the source reference object. Because this contravenes the rule for casting, which specifies that the target object should always be more generic than the source object, the syntax checker will complain that the source object can't be converted to the target object if you try to assign the objects using the `=` operator or use the `MOVE` statement. To bypass this restriction and tell the compiler that we know what we're doing, we use the `?=` operator for a widening cast, as shown:

```
Child ?= parent.
```

Using the `?=` operator will only bypass the check statically during compile time and postpones it until runtime. It will throw a runtime error if the target object is not more generic than the source object.

For example, the code in Listing 8.27 will pass the syntax checker (static check) but result in a runtime error because the child object is more specific than the parent object.

```
CLASS cl_parent DEFINITION.
  PUBLIC SECTION.
  METHODS meth1.
  METHODS meth2.
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
  METHOD meth1.
   WRITE 'In method1 of parent'.
  ENDMETHOD.
  METHOD meth2.
   WRITE 'In method2 of parent'.
  ENDMETHOD.
ENDCLASS.
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO cl_parent.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child.
Child ?= parent. "Results in runtime error
```

Listing 8.27 Widening Cast Resulting in Runtime Error

A widening cast should only be used when the dynamic type of the target object will be generic than the source object. This effectively means that a narrowing cast must be performed before a widening cast, as shown in Listing 8.28. Widening casts can be confusing and dangerous, so they should be used with care.

```
CLASS cl_parent DEFINITION.
  PUBLIC SECTION.
  METHODS meth1.
  METHODS meth2.
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
  METHOD meth1.
   WRITE 'In method1 of parent'.
  ENDMETHOD.
```



```
  METHOD meth2.
   WRITE 'In method2 of parent'.
  ENDMETHOD.
ENDCLASS.
CLASS cl_child1 DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3.
ENDCLASS.
CLASS cl_child1 IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child1'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child1'.
  ENDMETHOD.
ENDCLASS.
CLASS cl_child2 DEFINITION INHERITING FROM cl_child1.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3 REDEFINITION.
  METHODS meth4.
ENDCLASS.
CLASS cl_child2 IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child2'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child2'.
  ENDMETHOD.
  METHOD meth4.
     WRITE 'In method4 of child2'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO cl_parent.
DATA child1 TYPE REF TO cl_child1.
DATA child2 TYPE REF TO cl_child2.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child1.
CREATE OBJECT child2.
parent = child2.
child1 ?= parent.

child1->meth2( ).
```

Listing 8.28 Widening Cast Example

In Listing 8.28, three classes are defined: `cl_parent`, `cl_child1`, and `cl_child2`. `cl_child1` inherits from `cl_parent`, and `Cl_child2` inherits from `cl_child1`.

A narrow cast is first performed by assigning the `child2` object to the `parent` object, and then a widening cast is performed by assigning the `parent` object to `child1`. When the `meth2` method is called using the `child1` object reference after the widening cast, it will execute the code in the `meth2` method of the `cl_child2` class.

8.4.3 Dynamic Binding with the CALL Method

When an object is passed to the method as a parameter, the system automatically binds the object dynamically. This allows us to design objects that are easily extendable.

To understand dynamic binding with the `CALL` method, let's look at the code snippet in Listing 8.29. Here, we've defined `cl_student` as an abstract class with `tuition_fee` and `get_fee` as abstract methods with public visibility. A `fee_paid` attribute that can be set if the fee is paid by the student is defined in the protected section.

Two classes, `cl_commerce_student` and `cl_science_student`, inherit the `cl_student` class and maintain their own implementation of the `tuition_fee` and `get_fee` abstract methods. In the `tuition_fee` method, the private `fee_paid` attribute is set to `abap_true` if the student has paid the fee. The `get_fee` method returns the value of the private `fee_paid` attribute.

We've defined one `cl_admission` class that can be used to enroll a student in a course. This class has two methods, `set_student` and `enroll`, which can be used to complete the admission of the student in a course. Because this `cl_admission` class should be able to enroll any student from any student category (science student, commerce student, etc.), we need to design this class to import information about any student category.

The `set_student` method of the `cl_admission` class is defined with an importing parameter to import the `student` object. If we maintain the static type of this importing parameter as `cl_commerce_student`, it can only import the commerce student object and our class can only process the admission for commerce students. Similarly, if we maintain the static type of the importing parameter as `cl_science_student`, it can only import the science student object and our class can

only process the admission for science students. In order to make it work for any student category, we've maintained the importing parameter of this method as a static type of the `cl_student` abstract class, from which the `cl_science_student` and `cl_commerce_student` classes inherit.

```
CLASS cl_student DEFINITION ABSTRACT.
  PUBLIC SECTION.
  METHODS tuition_fee ABSTRACT.
  METHODS get_fee ABSTRACT RETURNING VALUE(fee_paid) TYPE boolean.
  PROTECTED SECTION.
  DATA fee_paid TYPE boolean.
ENDCLASS.
CLASS cl_commerce_student DEFINITION INHERITING FROM cl_student.
PUBLIC SECTION.
METHODS tuition_fee REDEFINITION.
METHODS get_fee REDEFINITION.
ENDCLASS.
CLASS cl_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
  "logic to calculate tuition fee for commerce students goes here
  "IF fee paid.
    fee_paid = abap_true.
  ENDMETHOD.
  METHOD get_fee.
    fee_paid = me->fee_paid.
  ENDMETHOD.
ENDCLASS.
CLASS cl_science_student DEFINITION INHERITING FROM cl_student.
  PUBLIC SECTION.
  METHODS tuition_fee REDEFINITION.
  METHODS get_fee REDEFINITION.
ENDCLASS.
CLASS cl_science_student IMPLEMENTATION.
  METHOD tuition_fee.
  "logic to calculate tuition fee for science students goes here
  "IF fee paid.
    fee_paid = abap_true.
  ENDMETHOD.
  METHOD get_fee.
    fee_paid = me->fee_paid.
  ENDMETHOD.
ENDCLASS.
CLASS cl_admission DEFINITION.
  PUBLIC SECTION.
  METHODS set_student IMPORTING im_student TYPE REF TO cl_student.
  METHODS enroll.
  PRIVATE SECTION.
  DATA admit TYPE boolean.
ENDCLASS.
```

```
CLASS cl_admission IMPLEMENTATION.
  METHOD set_student.
    IF im_student->get_fee( ) EQ abap_true.
    admit = abap_true.
    ENDIF.
  ENDMETHOD.
  METHOD enroll.
    IF admit EQ abap_true.
*Perform the steps to enroll
    ENDIF.
  ENDMETHOD.
ENDCLASS.

DATA : commerce_student TYPE REF TO cl_commerce_student,
       science_student TYPE REF TO cl_science_student,
       admission TYPE REF TO cl_admission.
START-OF-SELECTION.
CREATE OBJECT: commerce_student,
               science_student,
               admission.
CALL METHOD commerce_student->tuition_fee.
CALL METHOD admission->set_student( EXPORTING im_student = commerce_
student ).
CALL METHOD admission->enroll.

CALL METHOD science_student->tuition_fee.
CALL METHOD admission->set_student( EXPORTING im_student = science_
student ).
CALL METHOD admission->enroll.
```

Listing 8.29 Code for Dynamic Call Method Binding

When the `set_student` method of the `cl_admission` class is called, it allows us to pass any student object that is inherited from `cl_student`. Within this method, we're checking if the student has paid the fee by calling the `get_fee` method of the imported object and accordingly processing the admission of the student. Here, even though the static type of the imported object is different from the static type of the importing parameter, the system automatically binds the object to the correct instance.

With this design, we can easily process admissions for any future student category via the `cl_admission` class. All the student category class needs to do is to inherit the abstract class `cl_student`.

8.4.4 Interfaces

Similar to Java, ABAP Objects only supports single inheritance and don't support multiple inheritance as in C++. *Single inheritance* means a class can have multiple subclasses but only one superclass. Many subclasses can use the same class as their superclass, but each subclass can only have a single superclass.

Modern programming languages didn't add support for multiple inheritance (having multiple superclasses) in order to avoid ambiguity. One of the common problems with multiple inheritance is called the *diamond problem*. For example, as depicted in Figure 8.12, assume you have two subclasses `B` and `C` inheriting from the same superclass `A` and that both classes `B` and `C` have redefined a method `XYZ` of class `A` in their implementation. Now, if a new subclass `D` is defined as having both class `B` and class `C` as its superclasses then which superclass method will it use?

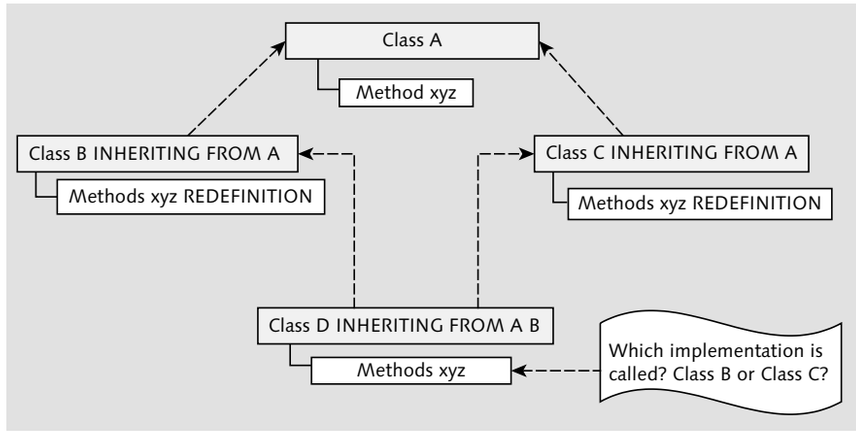


Figure 8.12 Diamond Problem with Multiple Inheritance

One way of implementing multiple inheritance is to use interfaces. Interfaces provide all the advantages of multiple inheritance while avoiding all the problems.

Interfaces are defined similarly to classes via the `INTERFACE` statement. An interface only contains a definition for which the implementation is maintained in the class that uses the interface. If you need a common definition in multiple classes, then you can create an interface and use it in multiple classes.

The syntax to define an interface is shown in Listing 8.30, in which interface `if_student` is defined using the `INTERFACE` statement and ended with the `ENDINTERFACE` statement.

```
INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.

CLASS cl_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_student.
ENDCLASS.
CLASS cl_science IMPLEMENTATION.
  METHOD if_student~meth1.
  ENDMETHOD.
ENDCLASS.
```

Listing 8.30 Interfaces

An interface can have all the components that a class supports, such as methods, attributes, events, and so on. All of the components are in `PUBLIC SECTION` by default, and you can't use any visibility section in the interface definition. Because an interface is generally created to be used in multiple classes, it makes sense to use the interface to define the public interface of the class, with each class that uses the interface implementing its own private or protected components.

In Listing 8.30, a `cl_science` class implements the `if_student` interface via the `INTERFACES` statement. In the implementation of the `cl_science` class, the `meth1` method defined in the `if_student` interface is implemented. Because the `meth1` method is defined in the interface and not in the class itself, the method is accessed using the `interface_name~component_name` syntax. Here, the tilde symbol (~) is the interface component selector that should be used to access the component of an interface.

Unlike classes, interfaces do not have any implementation section. Interfaces can be defined only in the global declaration area of a program and can't be defined within a class or a procedure or an event block.

Sometimes, the interface name can be too long. To avoid addressing the interface component using the long interface name, we can use aliases to make it easy while coding, as shown in Listing 8.31.


```
INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.
CLASS cl_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_student.
  ALIASES m1 FOR if_student~meth1.
ENDCLASS.
CLASS cl_science IMPLEMENTATION.
  METHOD m1.
  ENDMETHOD.
ENDCLASS.
DATA science TYPE REF TO cl_science.
START-OF-SELECTION.
CREATE OBJECT science.
science->m1( ).
```

Listing 8.31 Using Aliases

Interfaces can be nested by using the `INTERFACES` statement to include an interface within an interface. In Listing 8.32, the `if_student` interface is nested inside `if_college`. An alias is defined for the `meth1` method of `if_student` within `if_college`. The `cl_science` class implements the `if_college` interface. The `meth1` method of the `if_student` interface is accessed within the class using the alias defined for it in `if_college`.

```
INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.
INTERFACE if_college.
  INTERFACES if_student.
  ALIASES m1 FOR if_student~meth1.
  METHODS meth1.
ENDINTERFACE.
CLASS cl_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_college.
  ALIASES m1 FOR if_college~m1.
  ALIASES m2 FOR if_college~meth1.
ENDCLASS.
CLASS cl_science IMPLEMENTATION.
  METHOD m1.
  ENDMETHOD.
```

```
  METHOD m2.
  ENDMETHOD.
ENDCLASS.
DATA science TYPE REF TO cl_science.
START-OF-SELECTION.
CREATE OBJECT science.
science->m1( ).
```

Listing 8.32 Nesting Interfaces

The process of defining a global interface in Class Builder is similar to creating a class. Follow these steps to create an interface in Class Builder:

1. On the initial Class Builder screen (Transaction SE24; see Figure 8.13), enter the name of the interface and click the CREATE button. If the name starts with *ZIF*, Class Builder will automatically create an interface; otherwise, you'll see a dialog box (see Figure 8.14) to choose whether to create a class or interface.

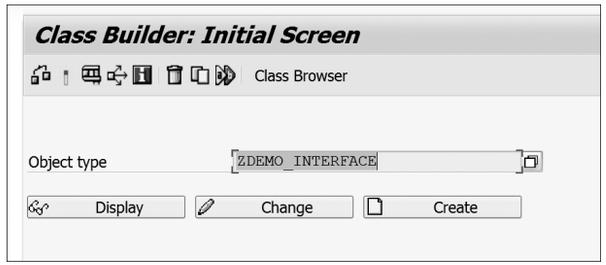


Figure 8.13 Class Builder: Initial Screen

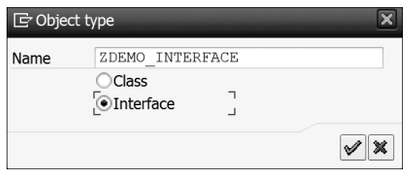


Figure 8.14 Object Type Selection Dialog Box

2. On the CLASS BUILDER: CHANGE INTERFACE... screen (see Figure 8.15), maintain the components of the interface and click the ACTIVATE button in the application toolbar to activate.

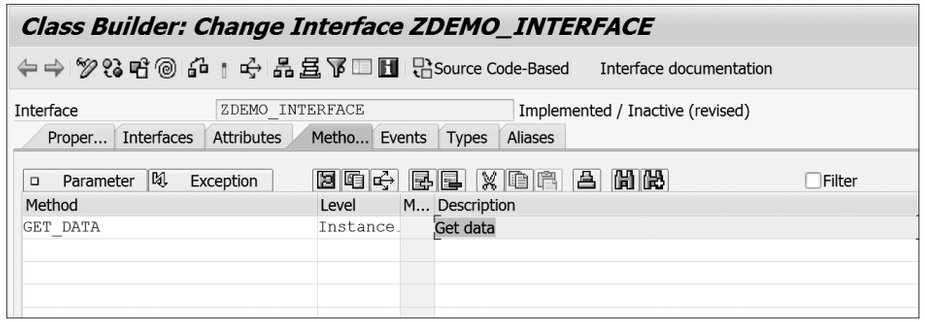


Figure 8.15 Class Builder: Change Interface Screen

3. You can include the interface in a global class under the INTERFACES tab, as shown in Figure 8.16.

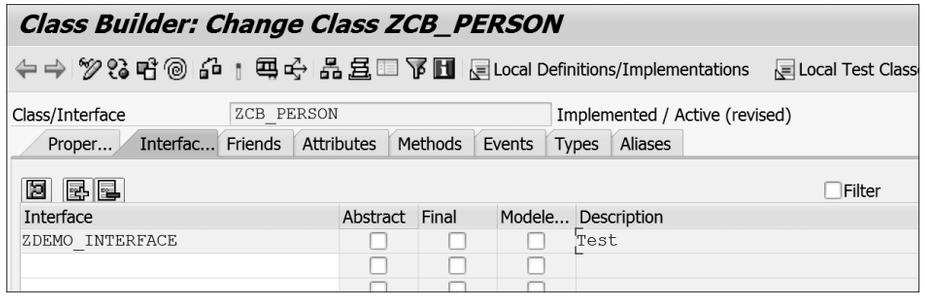


Figure 8.16 Including Interface in Global Class

4. The alias for an interface component can be maintained under the ALIASES tab, as shown in Figure 8.17. The name of the alias is maintained under the ALIAS column, and the visibility for the alias can be set under the VISIBLE column.

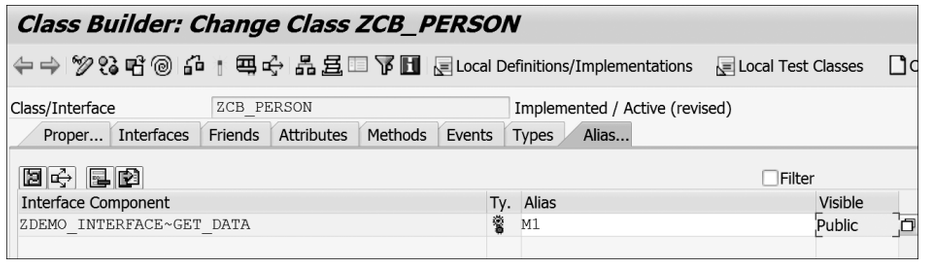


Figure 8.17 Aliases for Interface Component

In this section, we discussed the concept of polymorphism in OOP. In the next section, we'll take a bit of a departure from object-oriented concepts and look at how to work with XML.

8.5 Working with XML

Extensible Markup Language (XML) is a meta markup language that is used to define structured documents that can be easily exchanged between heterogeneous systems. There are many ways to exchange data between systems, but with the growth of web services, XML has gained popularity among developers. The beauty of XML lies in its flexibility and simplicity.

XML defines a standard that can be used to define the format of documents. Using XML, we can structure and organize various kind of data. A markup language (e.g., HTML) consists of *tags*, which are predefined. Tags allow you to format and organize the data. Tags are maintained between less than (<) and greater than (>) symbols. For example, in an HTML document, we can open the paragraph tag with `<p>` and close it with `</p>`. The content maintained between the tags is interpreted accordingly. Here, the content maintained between `<p>` and `</p>` is formatted as a paragraph; the content maintained between `<h1>` and `</h1>` is interpreted as a header.

However, XML is a *meta* markup language, which defines a markup language. It's designed to be flexible and easy to understand for both humans and machines. In XML, there are no predefined tags; any tag can be used to describe content.

Even though an in-depth understanding of XML documents and their processing is beyond the scope of this book, in this section we'll discuss the basic XML syntax and look at the iXML library provided by SAP to create XML documents in ABAP. Understanding XML is helpful when exposing or consuming data with external applications. This section should help you appreciate how OOP aids in designing libraries that are easy for developers to use in their programs.

8.5.1 XML Overview

XML files provide great flexibility to share structured documents. Because there are no predefined tags, we can use tags that suit our domain and that are self-

explanatory. Of course, developing a standard for document markup makes it easy for third-party vendors to develop software that can process XML files.

For example, say we have a website that can be extended by installing custom extensions. The extensions are of two types: components and plugins (i.e., the website can be extended by developing a component or a plugin). We've developed a component and want to upload the files of the custom component as a package to the webserver. To provide the information about the component to the installer script, we can supply an XML file with the package that contains information about the file structure of the package, as shown in Listing 8.33. The installer can read this XML file and upload the files in the package to the designated folders in the web server.

```
<?xml version="1.0" encoding="utf-8"?>
<!—This is comment-->
<extension type="component">
  <files folder="site">
        <filename>index.html</filename>
        <filename>site.php</filename>
  </files>
  <media folder="media">
    <folder>css</folder>
    <folder>images</folder>
    <folder>js</folder>
    </media>
</extension>
```

Listing 8.33 Sample XML Document

As shown in Listing 8.33, XML documents are organized into a series of elements. The first line in the document specifies the XML version and the encoding used. The syntax for this statement is `<?xml version="1.0" encoding=""?>` and it's optional.

The basic syntax to define an element in XML is as follows:

```
 <element_name attribute_name=attribute_value>
    <!-- Element Content -->
</element_name>
```

In the example XML file provided, for the `<extension type="component">` tag, the element name is `extension`, the attribute name is `type`, and the attribute value is `component`. This tells the upload script that we're trying to upload an extension that's a component. The tags `<files>` and `<media>` are the elements under the

parent node `<extension>`. The tags under `<files>` and `<media>` are the respective elements' content, which specifies the folders and files that we are uploading. The document is closed with the closing tag `</extension>`. Comments can be maintained between`<!--` and `-->`.

XML markup is case-sensitive; for example, the element name `<ELEMENT>` is different than `<element>`.

8.5.2 XML Processing Concepts

SAP NetWeaver AS ABAP provides an iXML library that can be used to process XML files. This library implements various interfaces that allow us to work with XML files by calling various methods.

Listing 8.34 shows the code to create an XML file using the iXML library. The XML file we're creating contains the data from Listing 8.33. In Listing 8.34, we're defining reference objects for the XML document from the iXML library. The reference object for the XML document is referenced to the `if_ixml_document` interface, and a reference object is defined for each element in the XML file by referencing `if_ixml_element`. The XML file is generated by calling the respective methods from the library, as shown in Listing 8.34.

```
REPORT ZDEMO_XML.
*Declarations to create XML document
DATA: lr_ixml TYPE REF TO if_ixml. "Reference for iXML object
"Reference for XML document
DATA: lr_document TYPE REF TO if_ixml_document.
."Reference for "extension" element in document
DATA: lr_extension  TYPE REF TO if_ixml_element

."Reference for "files" element in document
DATA: lr_files TYPE REF TO if_ixml_element
."Reference for "media" element in document
DATA: lr_media TYPE REF TO if_ixml_element
."Reference to set encoding
DATA: lr_encoding TYPE REF TO if_ixml_encoding

*Declarations to create output stream and render the file to
*application server directory
DATA: lr_streamfactory  TYPE REF TO if_ixml_stream_factory,
      lr_ostream        TYPE REF TO if_ixml_ostream,
      lr_renderer       TYPE REF TO if_ixml_renderer.
DATA  file_path TYPE string VALUE 'D:\USR\SAP\PUT\MANIFEST.XML'.
```



```
* Create iXML object
lr_ixml = cl_ixml=>create( ).
*Create Document
lr_document = lr_ixml->create_document( ).
*Create encoding
lr_encoding = lr_ixml->create_encoding( BYTE_ORDER = 0 CHARACTER_SET =
 'UTF-8').
*Set encoding
lr_document->set_encoding( lr_encoding ).
*Create element "extension" as root
lr_extension  = lr_document->create_simple_element(
name    = 'extension'
parent  = lr_document ).
*Set attribute for the "extension" element
lr_extension->set_attribute(           name  = 'Type'
VALUE = 'Component' ).
*Create "files" element with "extension" element as parent
lr_files = lr_document->create_simple_element(
name    = 'files'
parent  = lr_extension ).
*Set attribute for "files" element
lr_files->set_attribute(           name  = 'Folder'
VALUE = 'site' ).
*Create element content
lr_document->create_simple_element( name    = 'filename'
parent  = lr_files
VALUE   = 'index.html' ).
lr_document->create_simple_element( name    = 'filename'
parent  = lr_files
VALUE   = 'site.php' ).

*Create "media" element with "extension" element as parent
lr_media = lr_document->create_simple_element(
name    = 'media'
parent  = lr_extension ).
**Set attribute for "media" element
lr_media->set_attribute(           name  = 'Folder'
VALUE = 'media' ).

*Create element content
lr_document->create_simple_element( name    = 'folder'
parent  = lr_media
VALUE   = 'css' ).
lr_document->create_simple_element( name    = 'folder'
parent  = lr_media
VALUE   = 'images' ).
lr_document->create_simple_element( name    = 'folder'
parent  = lr_media
VALUE   = 'js' ).
```

```
* Create stream factory
lr_streamfactory = lr_ixml->create_stream_factory( ).
* Create output stream
lr_ostream  = lr_streamfactory->create_ostream_uri( system_id = file_
path ).
* Create renderer
lr_renderer = lr_ixml->create_renderer( ostream  = lr_ostream
document = lr_document ).
* Set pretty print
lr_ostream->set_pretty_print( abap_true ).
* Renders the attached document into output stream
lr_renderer->render( ).
```

Listing 8.34 Using iXML library to Create XML File

The code in Listing 8.34 saves the XML file in the application server directory, as shown in Figure 8.18.

SAP Directories

```
Directory:    D:\usr\sap\put
Name:         MANIFEST.XML
```

```
<?xml version="1.0" encoding="utf-8"?>
<extension Type="Component">
 <files Folder="site">
  <filename>index.html</filename>
  <filename>site.php</filename>
 </files>
 <media Folder="media">
  <folder>css</folder>
  <folder>images</folder>
  <folder>js</folder>
 </media>
</extension>
```

Figure 8.18 XML File

8.6 Summary

In this chapter, we explored various OOP concepts. The four pillars of OOP are encapsulation, inheritance, polymorphism, and abstraction. We discussed all of these concepts and looked at the various ways they're implemented in ABAP.

By now, you should have a good overview of objects and should feel comfortable working with ABAP Objects. Designing software using ABAP Objects takes some experience, but this chapter should serve as a starting point for you to start looking at designing applications by using OOP concepts.

In the next chapter, we'll use some of the concepts from this chapter to shed light on how to handle error situations in applications.

Contents

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

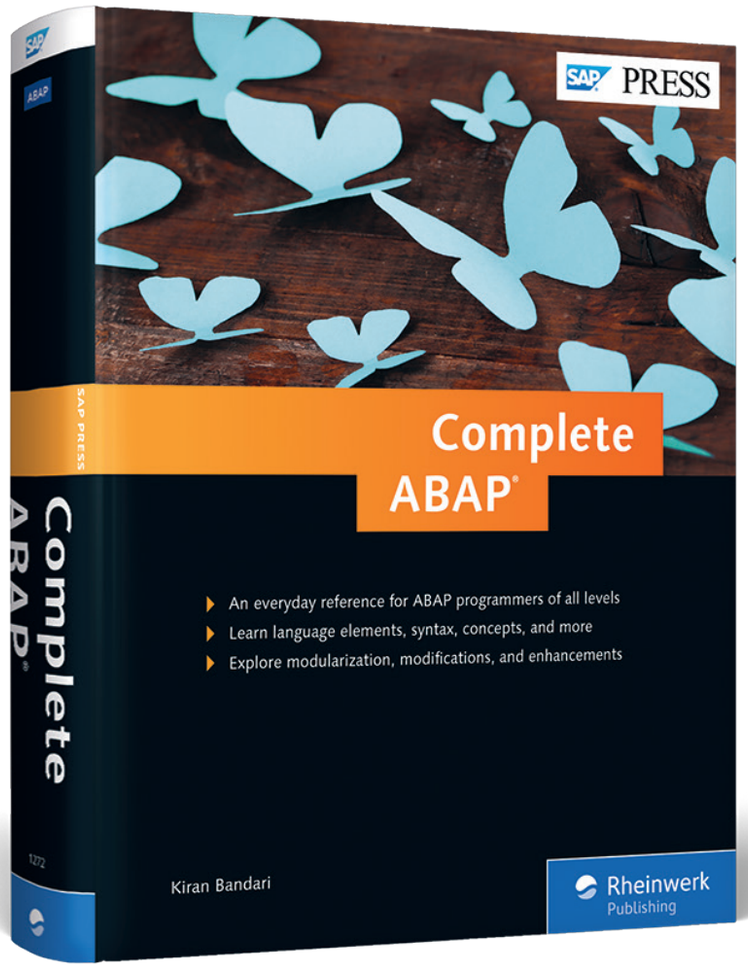
T

U

V

W

X



Kiran Bandari is a solution architect for one of the world's leading confection companies, and has been working with ABAP for more than 10 years. He has worked as a lead ABAP consultant on multiple SAP implementations, roll outs, and upgrade projects with a specific focus on custom development using ABAP Objects and Web Dynpro ABAP. He is also an industry trainer and has conducted ABAP training workshops for major clients like Wrigley's, IBM, Accenture, CapGemini, and more.

Kiran Bandari
Complete ABAP
1047 Pages, 2016, $79.95
ISBN 978-1-4932-1272-9

www.sap-press.com/3921