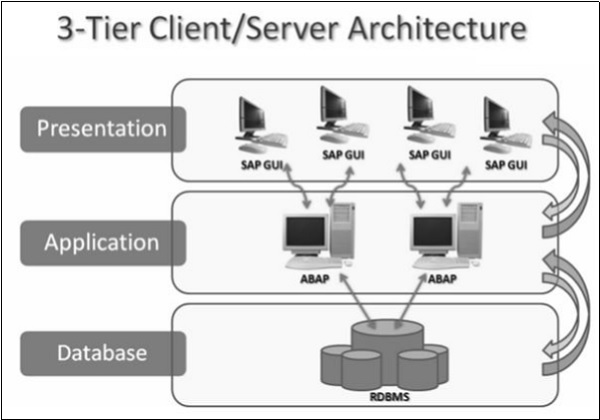
ABAP (Advanced Business Application Programming), is a fourth-generation programming language, used for development and customization purposes in the SAP software. Currently positioned along with Java, as the main language for SAP application server programming, most of the programs are executed under the control of the run-time system.

ABAP stands for Advanced Business Application Programming, a 4GL (4th generation) language. Currently it is positioned, along with Java, as the main language for SAP application server programming.

Let's start with the high level architecture of SAP system. The 3-tier Client/Server architecture of a typical SAP system is depicted as follows.



Using the ABAP Editor

**Step 1** − Start the transaction SE38 to navigate to the ABAP Editor (discussed in the next chapter). Let's start creating a report that is one of the many ABAP objects.

**Step 2** − On the initial screen of the editor, specify the name of your report in the input field PROGRAM. You may specify the name as ZHELLO1. The preceding Z is important for the name. Z ensures that your report resides in the customer namespace.

The customer namespace includes all objects with the prefix Y or Z. It is always used when customers or partners create objects (like a report) to differentiate these objects from objects of SAP and to prevent name conflicts with objects.

**Step 3** − You may type the report name in lower case letters, but the editor will change it to upper case. So the names of ABAP objects are ‘Not’ case sensitive.

**Step 4** − After specifying the name of the report, click the CREATE button. A popup window ABAP: PROGRAM ATTRIBUTES will pop up and you will provide more information about your report.

**Step 5** − Choose "Executable Program" as the report type, enter the title “My First ABAP Report” and then select SAVE to continue. The CREATE OBJECT DIRECTORY ENTRY window will pop up next. Select the button LOCAL OBJECT and the popup will close.

You can complete your first report by entering the WRITE statement below the REPORT statement, so that the complete report contains just two lines as follows −

REPORT ZHELLO1.

WRITE 'Hello World'.

Starting the Report

We can use the keyboard (Ctrl + S) or the save icon (right hand side beside the command field) to save the report. ABAP development takes place in AS ABAP.

Starting the report is as simple as saving it. Click the ACTIVATION button (left hand side next to the start icon) and start the report by using the icon DIRECT PROCESSING or the F8 function key. The title "My First ABAP Report" along with the output “Hello World” is displayed as well. Here is the output −

My First ABAP Report

Hello World

As long as you do not activate a new report or activate a change to an existing report, it is not relevant to their users. This is important in a central development environment where you may work on objects that other developers use in their projects.

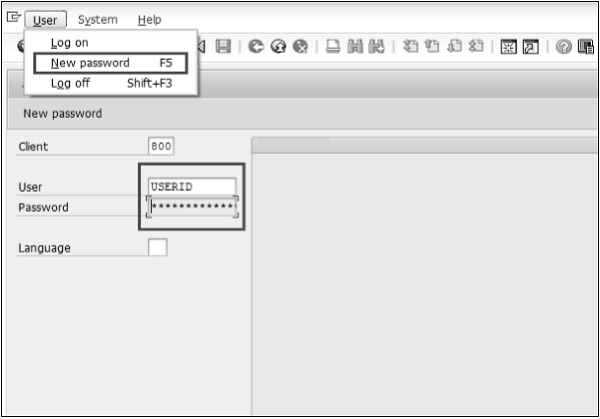
Viewing the Existing Code

If you look at the field Program and double-click on the value ZHELLO1, the ABAP editor will display the code for your report. This is called Forward Navigation. Double clicking on an object's name opens that object in the appropriate tool.

# Screen Navigation

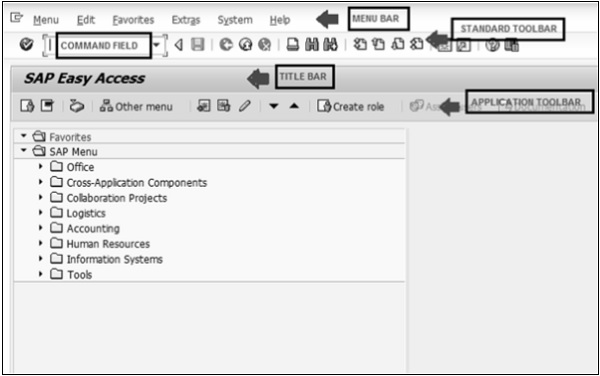
Login Screen

After you log on to SAP server, SAP login screen will prompt for User ID and Password. You need to provide a valid user ID and Password and press Enter (the user id and password is provided by system administrator). Following is the login screen.



Toolbar Icon

Following is the SAP screen toolbar.



**Menu Bar** − Menu bar is the top line of dialog window.

**Standard Toolbar** − Most standard functions such as Top of Page, End of Page, Page Up, Page Down and Save are available in this toolbar.

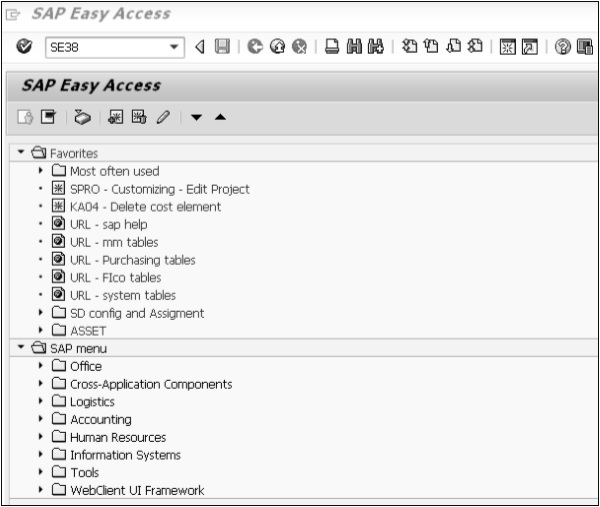
**Title Bar** − Title Bar displays the name of the application/business process you are currently in.

**Application Toolbar** − Application specific menu options are available here.

**Command Field** − We can start an application without navigating through the menu transactions and some logical codes are assigned to business processes. Transaction codes are entered in the command field to directly start the application.

ABAP Editor

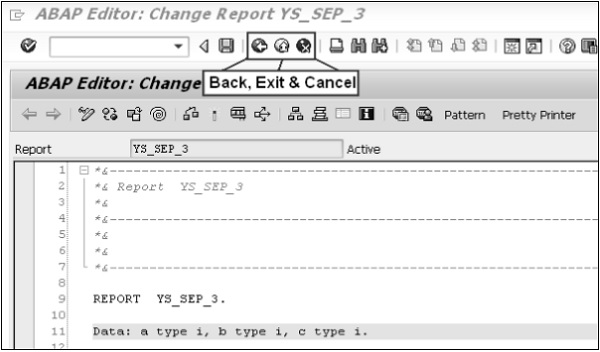
You may just start the transaction SE38 (enter SE38 in Command Field) to navigate to the ABAP Editor.



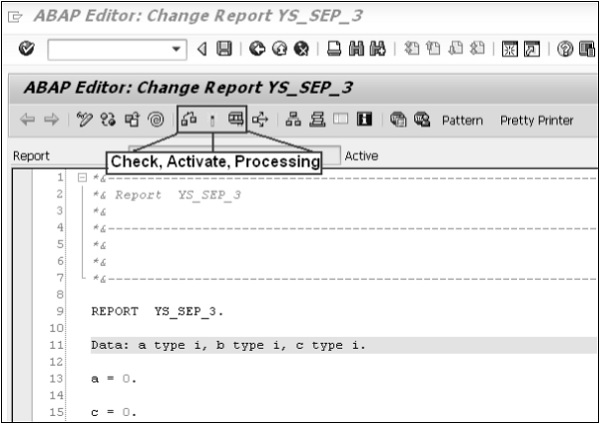
Standard Keys and Icons

**Exit keys** are used to exit the program/module or to log off. They are also used to go back to the last accessed screen.

Following are the standard exit keys used in SAP as shown in the image.

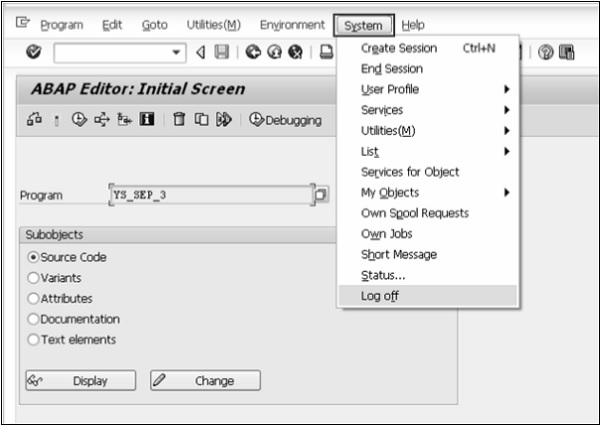


Following are the options for checking, activating and processing the reports.



Log Off

It’s always a good practice to Exit from your ABAP Editor or/and logoff from the SAP system after finishing your work.



# Basic Syntax

The syntax is −

REPORT [Program\_Name].

[Statements…].

This allows the statement to take up as many lines in the editor as it needs. For example, the REPORT may look like this −

REPORT Z\_Test123\_01.

Statements consist of a command and any variables and options, ending with a period. As long as the period appears at the end of the statement, no problems will arise. It is this period that marks where the statement finishes.

Let’s write the code.

On the line below the REPORT statement, just type this statement: Write ‘ABAP Tutorial’.

REPORT Z\_Test123\_01.

Write 'This is ABAP Tutorial'.

**Four things to consider while writing statements** −

* The write statement writes whatever is in quotes to the output window.
* The ABAP editor converts all text to uppercase except text strings, which are surrounded by single quotation marks.
* Unlike some older programming languages, ABAP does not care where a statement begins on a line. You may take advantage of this and improve the readability of your program by using indentation to indicate blocks of code.
* ABAP has no restrictions on the layout of statements. That is, multiple statements can be placed on a single line, or a single statement may stretch across multiple lines.

## Colon Notation

Consecutive statements can be chained together if the beginning of each statement is identical. This is done with the colon (:) operator and commas, which are used to terminate the individual statements, much as periods end normal statements.

Following is an example of a program that could save some key stroking −

WRITE 'Hello'.

WRITE 'ABAP'.

WRITE 'World'.

Using the colon notation, it could be rewritten this way −

WRITE: 'Hello',

'ABAP',

'World'.

Like any other ABAP statement, the layout doesn’t matter. This is an equally correct statement −

WRITE: 'Hello', 'ABAP', 'World'.

## Comments

Inline comments may be declared anywhere in a program by one of the two methods −

* Full line comments are indicated by placing an asterisk (\*) in the first position of the line, in which case the entire line is considered by the system to be a comment. Comments don’t need to be terminated by a period because they may not extend across more than one line −

\* This is the comment line

* Partial line comments are indicated by entering a double quote (") after a statement. All text following the double quote is considered by the system to be a comment. You need not terminate partial line comments by a period because they may not extend across more than one line −

WRITE 'Hello'. "Here is the partial comment

**Note** − Commented code is not capitalized by the ABAP editor.

## Suppressing Blanks

The NO-ZERO command follows the DATA statement. It suppresses all leading zeros of a number field containing blanks. The output is usually easier for the users to read.

### Example

REPORT Z\_Test123\_01.

DATA: W\_NUR(10) TYPE N.

MOVE 50 TO W\_NUR.

WRITE W\_NUR NO-ZERO.

The above code produces the following output −

50

**Note** − Without NO-ZERO command, the output is: 0000000050

## Blank Lines

The SKIP command helps in inserting blank lines on the page.

### Example

The message command is as follows −

WRITE 'This is the 1st line'.

SKIP.

WRITE 'This is the 2nd line'.

The above message command produces the following output −

This is the 1st line

This is the 2nd line

We may use the SKIP command to insert multiple blank lines.

SKIP number\_of\_lines.

The output would be several blank lines defined by the number of lines. The SKIP command can also position the cursor on a desired line on the page.

SKIP TO LINE line\_number.

This command is used to dynamically move the cursor up and down the page. Usually, a WRITE statement occurs after this command to put output on that desired line.

## Inserting Lines

The ULINE command automatically inserts a horizontal line across the output. It’s also possible to control the position and length of the line. The syntax is pretty simple −

ULINE.

### Example

The message command is as follows −

WRITE 'This is Underlined'.

ULINE.

The above code produces the following output −

This is Underlined (and a horizontal line below this).

## Messages

The MESSAGE command displays messages defined by a message ID specified in the REPORT statement at the beginning of the program. The message ID is a 2 character code that defines which set of 1,000 messages the program will access when the MESSAGE command is used.

The messages are numbered from 000 to 999. Associated with each number is a message text up to a maximum of 80 characters. When message number is called, the corresponding text is displayed.

Following are the characters for use with the Message command −

|  |  |  |
| --- | --- | --- |
| **Message** | **Type** | **Consequences** |
| E | Error | The message appears and the application halts at its current point. If the program is running in background mode, the job is canceled and the message is recorded in the job log. |
| W | Warning | The message appears and the user must press Enter for the application to continue. In background mode, the message is recorded in the job log. |
| I | Information | A pop-up window opens with the message text and the user must press Enter to continue. In background mode, the message is recorded in the job log. |
| A | Abend | This message class cancels the transaction that the user is currently using. |
| S | Success | This provides an informational message at the bottom of the screen. The information displayed is positive in nature and it is just meant for user feedback. The message does not impede the program in any way. |
| X | Abort | This message aborts the program and generates an ABAP short dump. |

Error messages are normally used to stop users from doing things they are not supposed to do. Warning messages are generally used to remind the users of the consequences of their actions. Information messages give the users useful information.

### Example

When we create a message for message the ID AB, the MESSAGE command - MESSAGE E011 gives the following output −

EAB011 This report does not support sub-number summarization.

# ABAP - Data Types

While programming in ABAP, we need to use a variety of variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. You may like to store information of various data types like character, integer, floating point, etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

## Elementary Data Types

ABAP offers the programmer a rich assortment of fixed length as well as variable length data types. Following table lists down ABAP elementary data types −

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Byte field | X |
| Text field | C |
| Integer | I |
| Floating point | F |
| Packed number | P |
| Text string | STRING |

Some of the fields and numbers can be modified using one or more names as the following −

* byte
* numeric
* character-like

The following table shows the data type, how much memory it takes to store the value in memory, and the minimum and maximum value that could be stored in such type of variables.

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Length** | **Typical Range** |
| X | 1 byte | Any byte values (00 to FF) |
| C | 1 character | 1 to 65535 |
| N (numeric text filed) | 1 character | 1 to 65535 |
| D (character-like date) | 8 characters | 8 characters |
| T (character-like time) | 6 characters | 6 characters |
| I | 4 bytes | -2147483648 to 2147483647 |
| F | 8 bytes | 2.2250738585072014E-308 to 1.7976931348623157E+308 positive or negative |
| P | 8 bytes | [-10^(2len -1) +1] to [+10^(2len -1) 1] (where len = fixed length) |
| STRING | Variable | Any alphanumeric characters |
| XSTRING (byte string) | Variable | Any byte values (00 to FF) |

### Example

REPORT YR\_SEP\_12.

DATA text\_line TYPE C LENGTH 40.

text\_line = 'A Chapter on Data Types'.

Write text\_line.

DATA text\_string TYPE STRING.

text\_string = 'A Program in ABAP'.

Write / text\_string.

DATA d\_date TYPE D.

d\_date = SY-DATUM.

Write / d\_date.

In this example, we have a character string of type C with a predefined length 40. STRING is a data type that can be used for any character string of variable length (text strings). Type STRING data objects should generally be used for character-like content where fixed length is not important.

The above code produces the following output −

A Chapter on Data Types

A Program in ABAP

12092015

The DATE type is used for the storage of date information and can store eight digits as shown above.

## Complex and Reference Types

The complex types are classified into **Structure types** and **Table types**. In the structure types, elementary types and structures (i.e. structure embedded in a structure) are grouped together. You may consider only the grouping of elementary types. But you must be aware of the availability of nesting of structures.

When the elementary types are grouped together, the data item can be accessed as a grouped data item or the individual elementary type data items (structure fields) can be accessed. The table types are better known as arrays in other programming languages. **Arrays** can be simple or structure arrays. In ABAP, arrays are called internal tables and they can be declared and operated upon in many ways when compared to other programming languages. The following table shows the parameters according to which internal tables are characterized.

|  |  |
| --- | --- |
| **S.No.** | **Parameter & Description** |
| 1 | **Line or row type**  Row of an internal table can be of elementary, complex or reference type. |
| 2 | **Key**  Specifies a field or a group of fields as a key of an internal table that identifies the table rows. A key contains the fields of elementary types. |
| 3 | **Access method**  Describes how ABAP programs access individual table entries. |

# ABAP - Variables

Variables are named data objects used to store values within the allotted memory area of a program. As the name suggests, users can change the content of variables with the help of ABAP statements. Each variable in ABAP has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. The basic form of a variable declaration is −

DATA <f> TYPE <type> VALUE <val>.

Here <f> specifies the name of a variable. The name of the variable can be up to 30 characters long. <type> specifies the type of variable. Any data type with fully specified technical attributes is known as <type>. The <val> specifies the initial value of the of <f> variable. In case you define an elementary fixed-length variable, the DATA statement automatically populates the value of the variable with the type-specific initial value. Other possible values for <val> can be a literal, constant, or an explicit clause, such as Is INITIAL.

Following are valid examples of variable declarations.

DATA d1(2) TYPE C.

DATA d2 LIKE d1.

DATA minimum\_value TYPE I VALUE 10.

In the above code snippet, d1 is a variable of C type, d2 is a variable of d1 type, and minimum\_value is a variable of ABAP integer type I.

This chapter will explain various variable types available in ABAP. There are three kinds of variables in ABAP −

* Static Variables
* Reference Variables
* System Variables

## Static Variables

* Static variables are declared in subroutines, function modules, and static methods.
* The lifetime is linked to the context of the declaration.
* With ‘CLASS-DATA’ statement, you can declare variables within the classes.
* The ‘PARAMETERS’ statement can be used to declare the elementary data objects that are linked to input fields on a selection screen.
* You can also declare the internal tables that are linked to input fields on a selection screen by using ‘SELECT-OPTIONS’ statement.

Following are the conventions used while naming a variable −

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

This program shows how to declare a variable using the PARAMETERS statement −

REPORT ZTest123\_01.

PARAMETERS: NAME(10) TYPE C,

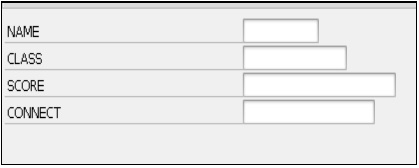
CLASS TYPE I,

SCORE TYPE P DECIMALS 2,

CONNECT TYPE MARA-MATNR.

Here, NAME represents a parameter of 10 characters, CLASS specifies a parameter of integer type with the default size in bytes, SCORE represents a packed type parameter with values up to two decimal places, and CONNECT refers to the MARA-MATNF type of ABAP Dictionary.

The above code produces the following output −



## Reference Variables

The syntax for declaring reference variables is −

DATA <ref> TYPE REF TO <type> VALUE IS INITIAL.

* REF TO addition declares a reference variable ref.
* The specification after REF TO specifies the static type of the reference variable.
* The static type restricts the set of objects to which <ref> can refer.
* The dynamic type of reference variable is the data type or class to which it currently refers.
* The static type is always more general or the same as the dynamic type.
* The TYPE addition is used to create a bound reference type and as a start value, and only IS INITIAL can be specified after the VALUE addition.

### Example

CLASS C1 DEFINITION.

PUBLIC SECTION.

DATA Bl TYPE I VALUE 1.

ENDCLASS. DATA: Oref TYPE REF TO C1 ,

Dref1 LIKE REF TO Oref,

Dref2 TYPE REF TO I .

CREATE OBJECT Oref.

GET REFERENCE OF Oref INTO Dref1.

CREATE DATA Dref2.

Dref2→\* = Dref1→\*→Bl.

* In the above code snippet, an object reference Oref and two data reference variables Dref1 and Dref2 are declared.
* Both data reference variables are fully typed and can be dereferenced using the dereferencing operator →\* at operand positions.

## System Variables

* ABAP system variables are accessible from all ABAP programs.
* These fields are actually filled by the run-time environment.
* The values in these fields indicate the state of the system at any given point of time.
* You can find the complete list of system variables in the SYST table in SAP.
* Individual fields of the SYST structure can be accessed by using either “SYST-” or “SY-”.

### Example

REPORT Z\_Test123\_01.

WRITE:/'SY-ABCDE', SY-ABCDE,

/'SY-DATUM', SY-DATUM,

/'SY-DBSYS', SY-DBSYS,

/'SY-HOST ', SY-HOST,

/'SY-LANGU', SY-LANGU,

/'SY-MANDT', SY-MANDT,

/'SY-OPSYS', SY-OPSYS,

/'SY-SAPRL', SY-SAPRL,

/'SY-SYSID', SY-SYSID,

/'SY-TCODE', SY-TCODE,

/'SY-UNAME', SY-UNAME,

/'SY-UZEIT', SY-UZEIT.

The above code produces the following output −

SY-ABCDE ABCDEFGHIJKLMNOPQRSTUVWXYZ

SY-DATUM 12.09.2015

SY-DBSYS ORACLE

SY-HOST sapserver

SY-LANGU EN

SY-MANDT 800

SY-OPSYS Windows NT

SY-SAPRL 700

SY-SYSID DMO

SY-TCODE SE38

SY-UNAME SAPUSER

SY-UZEIT 14:25:48

# ABAP - Constants & Literals

Literals are unnamed data objects that you create within the source code of a program. They are fully defined by their value. You can’t change the value of a literal. Constants are named data objects created statically by using declarative statements. A constant is declared by assigning a value to it that is stored in the program's memory area. The value assigned to a constant can’t be changed during the execution of the program. These fixed values can also be considered as literals. There are two types of literals − numeric and character.

## Numeric Literals

Number literals are sequences of digits which can have a prefixed sign. In number literals, there are no decimal separators and no notation with mantissa and exponent.

Following are some examples of numeric literals −

183.

-97.

+326.

## Character Literals

Character literals are sequences of alphanumeric characters in the source code of an ABAP program enclosed in single quotation marks. Character literals enclosed in quotation marks have the predefined ABAP type C and are described as text field literals. Literals enclosed in “back quotes” have the ABAP type STRING and are described as string literals. The field length is defined by the number of characters.

**Note** − In text field literals, trailing blanks are ignored, but in string literals they are taken into account.

Following are some examples of character literals.

### Text field literals

REPORT YR\_SEP\_12.

Write 'Tutorials Point'.

Write / 'ABAP Tutorial'.

### String field literals

REPORT YR\_SEP\_12.

Write `Tutorials Point `.

Write / `ABAP Tutorial `.

The output is same in both the above cases −

Tutorials Point

ABAP Tutorial

**Note** − When we try to change the value of the constant, a syntax or run-time error may occur. Constants that you declare in the declaration part of a class or an interface belong to the static attributes of that class or interface.

## CONSTANTS Statement

We can declare the named data objects with the help of CONSTANTS statement.

Following is the syntax −

CONSTANTS <f> TYPE <type> VALUE <val>.

The CONSTANTS statement is similar to the DATA statement.

<f> specifies a name for the constant. TYPE <type> represents a constant named <f>, which inherits the same technical attributes as the existing data type <type>. VALUE <val> assigns an initial value to the declared constant name <f>.

**Note** − We should use the VALUE clause in the CONSTANTS statement. The clause ‘VALUE’ is used to assign an initial value to the constant during its declaration.

We have 3 types of constants such as elementary, complex and reference constants. The following statement shows how to define constants by using the CONSTANTS statement −

REPORT YR\_SEP\_12.

CONSTANTS PQR TYPE P DECIMALS 4 VALUE '1.2356'.

Write: / 'The value of PQR is:', PQR.

The output is −

The value of PQR is: 1.2356

Here it refers to elementary data type and is known as elementary constant.

Following is an example for complex constants −

BEGIN OF EMPLOYEE,

Name(25) TYPE C VALUE 'Management Team',

Organization(40) TYPE C VALUE 'Tutorials Point Ltd',

Place(10) TYPE C VALUE 'India',

END OF EMPLOYEE.

In the above code snippet, EMPLOYEE is a complex constant that is composed of the Name, Organization and Place fields.

The following statement declares a constant reference −

CONSTANTS null\_pointer TYPE REF TO object VALUE IS INITIAL.

# SAP ABAP - Operators

ABAP provides a rich set of operators to manipulate variables. All ABAP operators are classified into four categories −

* Arithmetic Operators
* Comparison Operators
* Bitwise Operators
* Character String Operators

## Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following list describes arithmetic operators. Assume integer variable A holds 20 and variable B holds 40.

|  |  |
| --- | --- |
| **S.No.** | **Arithmetic Operator & Description** |
| 1 | **+ (Addition)**  Adds values on either side of the operator. Example: A + B will give 60. |
| 2 | **− (Subtraction)**  Subtracts right hand operand from left hand operand. Example: A − B will give -20. |
| 3 | **\* (Multiplication)**  Multiplies values on either side of the operator. Example: A \* B will give 800. |
| 4 | **/ (Division)**  Divides left hand operand by right hand operand. Example: B / A will give 2. |
| 5 | **MOD (Modulus)**  Divides left hand operand by right hand operand and returns the remainder. Example: B MOD A will give 0. |

### Example

REPORT YS\_SEP\_08.

DATA: A TYPE I VALUE 150,

B TYPE I VALUE 50,

Result TYPE I.

Result = A / B.

WRITE / Result.

The above code produces the following output −

3

## Comparison Operators

Let’s discuss the various types of comparison operators for different operands.

|  |  |
| --- | --- |
| **S.No.** | **Comparison Operator & Description** |
| 1 | **= (equality test). Alternate form is EQ.**  Checks if the values of two operands are equal or not, if yes then condition becomes true. Example (A = B) is not true. |
| 2 | **<> (Inequality test). Alternate form is NE.**  Checks if the values of two operands are equal or not. If the values are not equal then the condition becomes true. Example (A <> B) is true. |
| 3 | **> (Greater than test). Alternate form is GT.**  Checks if the value of left operand is greater than the value of right operand. If yes then condition becomes true. Example (A > B) is not true. |
| 4 | **< (Less than test). Alternate form is LT.**  Checks if the value of left operand is less than the value of right operand. If yes, then condition becomes true. Example (A < B) is true. |
| 5 | **>= (Greater than or equals) Alternate form is GE.**  Checks if the value of left operand is greater than or equal to the value of right Operand. If yes, then condition becomes true. Example (A >= B) is not true. |
| 6 | **<= (Less than or equals test). Alternate form is LE.**  Checks if the value of left operand is less than or equal to the value of right operand. If yes, then condition becomes true. Example (A <= B) is true. |
| 7 | **a1 BETWEEN a2 AND a3 (Interval test)**  Checks whether a1 lies in between a2 and a3 (inclusive). If yes, then the condition becomes true. Example (A BETWEEN B AND C) is true. |
| 8 | **IS INITIAL**  The condition becomes true if the contents of the variable have not changed and it has been automatically assigned its initial value. Example (A IS INITIAL) is not true |
| 9 | **IS NOT INITIAL**  The condition becomes true if the contents of the variable have changed. Example (A IS NOT INITIAL) is true. |

**Note** − If the data type or length of the variables does not match then automatic conversion is performed. Automatic type adjustment is performed for either one or both of the values while comparing two values of different data types. The conversion type is decided by the data type and the preference order of the data type.

Following is the order of preference −

* If one field is of type I, then the other is converted to type I.
* If one field is of type P, then the other is converted to type P.
* If one field is of type D, then the other is converted to type D. But C and N types are not converted and they are compared directly. Similar is the case with type T.
* If one field is of type N and the other is of type C or X, both the fields are converted to type P.
* If one field is of type C and the other is of type X, the X type is converted to type C.

### Example 1

REPORT YS\_SEP\_08.

DATA: A TYPE I VALUE 115,

B TYPE I VALUE 119.

IF A LT B.

WRITE: / 'A is less than B'.

ENDIF

The above code produces the following output −

A is less than B

### Example 2

REPORT YS\_SEP\_08.

DATA: A TYPE I.

IF A IS INITIAL.

WRITE: / 'A is assigned'.

ENDIF.

The above code produces the following output −

A is assigned.

## Bitwise Operators

ABAP also provides a series of bitwise logical operators that can be used to build Boolean algebraic expressions. The bitwise operators can be combined in complex expressions using parentheses and so on.

|  |  |
| --- | --- |
| **S.No.** | **Bitwise Operator & Description** |
| 1 | **BIT-NOT**  Unary operator that flips all the bits in a hexadecimal number to the opposite value. For instance, applying this operator to a hexadecimal number having the bit level value 10101010 (e.g. 'AA') would give 01010101. |
| 2 | **BIT-AND**  This binary operator compares each field bit by bit using the Boolean AND operator. |
| 3 | **BIT-XOR**  Binary operator that compares each field bit by bit using the Boolean XOR (exclusive OR) operator. |
| 4 | **BIT-OR**  Binary operator that compares each field bit by bit using the Boolean OR operator. |

For example, following is the truth table that shows the values generated when applying the Boolean AND, OR, or XOR operators against the two bit values contained in field A and field B.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Field A** | **Field B** | **AND** | **OR** | **XOR** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

## Character String Operators

Following is a list of character string operators −

|  |  |
| --- | --- |
| **S.No.** | **Character String Operator & Description** |
| 1 | **CO (Contains Only)**  Checks whether A is solely composed of the characters in B. |
| 2 | **CN (Not Contains ONLY)**  Checks whether A contains characters that are not in B. |
| 3 | **CA (Contains ANY)**  Checks whether A contains at least one character of B. |
| 4 | **NA (NOT Contains Any)**  Checks whether A does not contain any character of B. |
| 5 | **CS (Contains a String)**  Checks whether A contains the character string B. |
| 6 | **NS (NOT Contains a String)**  Checks whether A does not contain the character string B. |
| 7 | **CP (Contains a Pattern)**  It checks whether A contains the pattern in B. |
| 8 | **NP (NOT Contains a Pattern)**  It checks whether A does not contain the pattern in B. |

### Example

REPORT YS\_SEP\_08.

DATA: P(10) TYPE C VALUE 'APPLE',

Q(10) TYPE C VALUE 'CHAIR'.

IF P CA Q.

WRITE: / 'P contains at least one character of Q'.

ENDIF.

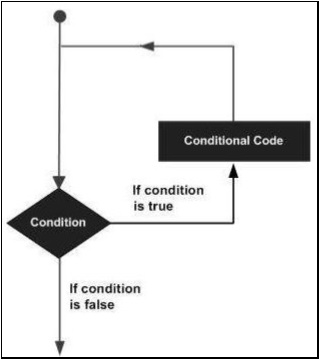
The above code produces the following output −

P contains at least one character of Q.

# ABAP - Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths. A **loop statement** allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages.



ABAP programming language provides the following types of loop to handle looping requirements.

|  |  |
| --- | --- |
| **S.No.** | **Loop Type & Description** |
| 1 | [**WHILE loop**](https://www.tutorialspoint.com/sap_abap/sap_abap_while_loop.htm)  Repeats a statement or group of statements when a given condition is true. It tests the condition before executing the loop body. |
| 2 | [**Do loop**](https://www.tutorialspoint.com/sap_abap/sap_abap_do_loop.htm)  The DO statement is useful for repeating particular task a specific number of times. |
| 3 | [**Nested loop**](https://www.tutorialspoint.com/sap_abap/sap_abap_nested_loop.htm)  You may use one or more loops inside any another WHILE or DO loop. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. ABAP includes control statements that allow loops to be ended prematurely. It supports the following control statements.

|  |  |
| --- | --- |
| **S.No.** | **Control Statement & Description** |
| 1 | [**CONTINUE**](https://www.tutorialspoint.com/sap_abap/sap_abap_continue_statement.htm)  Causes the loop to skip the remainder of its body and starts the next loop pass. |
| 2 | [**CHECK**](https://www.tutorialspoint.com/sap_abap/sap_abap_check_statement.htm)  If the condition is false, then the remaining statements after the CHECK are just ignored and the system starts the next loop pass. |
| 3 | [**EXIT**](https://www.tutorialspoint.com/sap_abap/sap_abap_exit_statement.htm)  Terminates the loop entirely and transfers execution to the statement immediately following the loop. |

# If Statement

The following syntax is used for the IF statement.

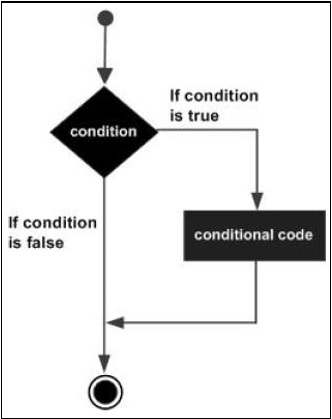
IF<condition\_1>.

<Statements...>.

ENDIF.

If the expression evaluates to true, then the IF block of code will be executed.

## Flow Diagram



## Example

Report YH\_SEP\_15.

Data Title\_1(20) TYPE C.

Title\_1 = 'Tutorials'.

IF Title\_1 = 'Tutorials'.

write 'This is IF statement'.

ENDIF.

The above code produces the following output −

This is IF statement.

In case of IF….ELSE statements, if the expression evaluates to true then the IF block of code will be executed. Otherwise, ELSE block of code will be executed.

The following syntax is used for IF….ELSE statement.

IF<condition\_1>.

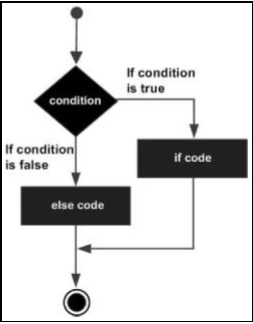
<statement block 1>.

ELSE.

<statement block 2>.

ENDIF.

## Flow Diagram



## Example

Report YH\_SEP\_15.

Data Title\_1(20) TYPE C.

Title\_1 = 'Tutorials'.

IF Title\_1 = 'Tutorial'.

write 'This is IF Statement'.

ELSE.

write 'This is ELSE Statement'.

ENDIF.

The above code produces the following output −

This is ELSE Statement.

## IF….ELSEIF….ELSE Statement

Sometimes nesting of the IF statements can make the code difficult to understand. In such cases, the ELSEIF statement is used to avoid nesting of the IF statement.

When using IF, ELSEIF and ELSE statements there are a few points to consider −

* An IF statement can have zero or one ELSE statement and it must come after any ELSEIF statement.
* An IF statement can have zero to many ELSEIF statements and they must come before the ELSE statement.
* If an ELSEIF statement succeeds, none of the remaining ELSEIF statements or ELSE statement will be tested.

The following syntax is used for the IF....ELSEIF….ELSE statement.

IF<condition\_1>.

<statement block 1>.

ELSEIF<condition\_2>.

<statement block 2>.

ELSEIF<condition\_3>.

<statement block 3>.

......

......

......

......

ELSE.

<statement block>.

ENDIF.

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

* condition\_1 of IF statement represents a logical condition that evaluates a true or false condition.
* condition\_2 shows the second condition specified in the ELSEIF statement, which is executed when the IF statement condition turns out to be false.
* ENDIF denotes the end of the IF statement block.

### Example

Report YH\_SEP\_15.

Data Result TYPE I VALUE 65.

IF Result < 0.

Write / 'Result is less than zero'.

ELSEIF Result < 70.

Write / 'Result is less than seventy'.

ELSE.

Write / 'Result is greater than seventy'.

ENDIF.

The above code produces the following output −

Result is less than seventy.