VERD PROGRAMMING LANGUAGE
REFERENCE MANUAL

TABLE OF CONTENTS

TABLE LIST

CODE SAMPLE LIST

1. Introduction

VERD is a programming language that was developed by using C++ and JavaScript languages. VERD is an abbreviation for “Veri Tabanı Raporlama Dili” in Turkish. “Veri Tabanı Raporlama Dili” in Turkish means “Database Reporting Language” in English. VERD language takes some features from ABAP, C and HTML. VERD is designed to ease ERP development and database reporting.

VERD language runs in either embedded or client/server mode. In embedded form client and server runs in a single application. Embedded form is used mainly for debugging purposes. Server is core of the language in client/server mode and server is written by using C++ language. Client and server communicate by exchanging XML data. Client can be written in any language (C++, Javascript, Python etc.)

Currently, C++ and Javascript (Web) Clients are avaliable. Most mature and tested client is C++ Client. C++ Client is written by using WxWidgets and Expat libraries. SlickGrid and Bootstrap javascript libraries are used for Web Client. VERD programming language currently supports only Firebird database management systems. In the future, it is planned to add support for SQL Server, PostgreSQL, MariaDB and Oracle database management systems. VERD programming language is cross-platform. It can run on both Windows and Linux. WxWidgets and Expat are chosen for that purpose, since they are cross-platform.

The purpose of this manual is to tell you how to use the features of the VERD programming language.

2. VERD Programming Language

2.1. Database Interface Component

Database interface component represents the relational database that compiler and runtime of VERD language uses. During compilation, compiler automatically connects to the database with given parameters and uses any table defined within database as a type.

Compiler does not create types for all database tables defined in database, but in any type definition with TYPE construct and in any data definition with DATA construct, it automatically checks whether given type exists in database and if yes it creates this database table as a type within program.

Runtime connects the database defined with database interface component during runtime initialization with given parameters. For SELECT, INSERT, UPDATE and DELETE constructs runtime performs operations on the defined database.

Compiler and runtime might connect different databases. While compiler can connect a template database that only holds table definitions, runtime database connects to production database. For convenience, table definitions for both databases should match. Compiler and runtime database information can be given in a single database interface component. Compiler and runtime connection information for a single database interface component are kept in separate fields.

A program can have multiple database interface components. But currently, specifying the database in SELECT, INSERT, UPDATE and DELETE constructs is not supported. So if multiple databases have the same table, then first one would be taken by the compiler as source in SELECT, INSERT, UPDATE and DELETE constructs. In the future it is planned to add support for specifying the database in SELECT, INSERT, UPDATE and DELETE constructs.

Currently only Firebird and SQL Server relation database management systems are supported by VERD language. In the future, it is planned to add support for PostgreSQL, MariaDB and Oracle database management systems.

The fields within the Database interface component are explained below along with their intended use (Table 2.1).

Table 2.1 *Database Interface Component*

Field Name	**Data Type**	**Purpose of use**
DBID	Short	It refers to the field where the id of the database interface component is kept.
NAME	Char(30)	It refers to the field where the name of the database interface component is kept.
DBNAME	Integer	It refers to field where the name of the database is kept.
DBTYPE	Char(30)	It refers to the type of database. Currently only Firebird and SQL Server RDMS systems are supported.
CSERVER	Char(30)	It refers to the field where ip address or network name of server that database that is used by compiler stays in.
CUSRNAME	Char(30)	It refers to the field where user name to connect database that is used by compiler.
CPASSWD	Char(20)	It refers to the field where password to connect database that is used by compiler.
CURL	Char(90)	It refers to the field where path of database within server for the database that is used by compiler.
RTSERVER	Char(30)	It refers to the field where ip address or network name of server that database that is used by runtime stays in.
RTUSRNAME	Char(30)	It refers to the field where user name to connect database that is used by runtime.
RTPASSWD	Char(20)	It refers to the field where password to connect database that is used by runtime.
RTURL	Char(90)	It refers to the field where path of database within server for the database that is used by runtime.
AUTOCOMMIT	Char(30)	It refers to the field whether the database operations would be committed automatically or by using "**commit**" statement.

2.2. User Interface Components

A detailed explanation of the user interface components used in the Database Reporting language is provided below.

2.2.1. **Frame**

A frame represents the highest-level user interface component. Almost all user interface components are created under the frame element. The display frame and refresh frame commands are used within the program to display the frame element and update its content.

The fields within the Frame user interface component are explained below along with their intended use (***Table 2.2***).

Table 2.2 *Frame User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Frame is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Frame. This field is used internally.
TOP	Short	It refers to the top starting point of the Frame within the screen.
LEFT	Short	It refers to the left starting point of the Frame within the screen.
WIDTH	Short	It refers to the width of the Frame on the screen.
HEIGHT	Short	It refers to the height of the Frame on the screen.
SWIDTH	Short	In case that the maximize field for the Frame is false, it refers to the screen width that will be taken into account for different screen resolutions..
SHEIGHT	Short	In case that the maximize field for the Frame is false, it refers to the screen height that will be taken into account for different screen resolutions..
SCALE	Bool	It refers to whether Frame would be automaticly resized at different screen resolutions.
MAXIMIZE	Bool	It refers whether the Frame covers the entire screen or not.
ON-INIT	Routine	It refers the routine to be called just before the Frame is displayed.

NOOFSUB	Short	It refers to the number of other user interface components defined under the Frame. This field is used internally.
SUBOBJ	Integer	It refers to the address of other user interface components defined under the Frame. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the Frame would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the Frame would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the Frame would be automatically recalculated when the screen is resized.

2.2.2. **Block**

A block is a user interface component that contains other interface components. Almost all user interface components can be created within a block interface component.

The fields within the block interface component are explained below along with their intended use (***Table 2.3***).

Table 2.3 *Block User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Block is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Block. This field is used internally.
TOP	Short	It refers to the top starting point of the Block within the screen.
LEFT	Short	It refers to the left starting point of the Block within the screen.
WIDTH	Short	It refers to the width of the Block on the screen.
HEIGHT	Short	It refers to the height of the Block on the screen.
NOOFSUB	Short	It refers to the number of other user interface components defined under the Block. This field is used internally.
SUBOBJ	Integer	It refers to the address of other user interface components defined under the Block. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the Block would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the Block would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the Block would be automatically recalculated when the screen is resized.

2.2.3. **TabControl**

A TabControl is a user interface component that contains one or more Tab interface components. Only Tab interface component can be created within a TabControl interface component.

The fields within the TabControl interface component are explained below along with their intended use (***Table 2.4***).

Table 2.4 *TabControl User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the TabControl is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the TabControl. This field is used internally.
TOP	Short	It refers to the top starting point of the TabControl within the screen.
LEFT	Short	It refers to the left starting point of the TabControl within the screen.
WIDTH	Short	It refers to the width of the TabControl on the screen.
HEIGHT	Short	It refers to the height of the TabControl on the screen.
NOOFSUB	Short	It refers to the number of Tab interface components defined under the TabControl.
SUBTAB	Integer	It refers to the address of the Tab interface components defined under the TabControl. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the TabControl would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the TabControl would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the TabControl would be automatically recalculated when the screen is resized.

2.2.4. **Tab**

A tab is a user interface component that contains other interface components. Almost all interface elements can be created within a tab interface component. A tab interface component can only be created within a TabControl interface component.

The fields within the Tab interface element are explained below along with their intended use (***Table 2.5***).

Table 2.5 *Tab User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Tab is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Tab. This field is used internally.
TITLE	Char(30)	It refers to the top starting point of the Tab within the screen.
TOP	Short	It refers to the left starting point of the Tab within the screen.
LEFT	Short	It refers to the width of the Tab on the screen.
WIDTH	Short	It refers to the height of the Tab on the screen.
HEIGHT	Short	It refers to the field where the name of the Tab is kept.
NOOFSUB	Short	It refers to the number of other user interface components defined under the Tab. This field is used internally.
SUBOBJ	Integer	It refers to the address of other user interface components defined under the Tab. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the Tab would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the Tab would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the Tab would be automatically recalculated when the screen is resized.

2.2.5. **SearchBlock**

SearchBlock refers to a user interface component that contains only the SearchOption and Grid interface components. A SearchBlock can contain multiple SearchOption interface components, but only one Grid interface component.

The fields within the SearchBlock interface element are explained below along with their intended use (***Table 2.6***).

Table 2.6 *SearchBlock User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the SearchBlock is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the SearchBlock. This field is used internally.
TOP	Short	It refers to the top starting point of the SearchBlock within the screen.
LEFT	Short	It refers to the left starting point of the SearchBlock within the screen.
WIDTH	Short	It refers to the width of the SearchBlock on the screen.
HEIGHT	Short	It refers to the height of the SearchBlock on the screen.
NOOFSUB	Short	It refers to the number of other user interface components defined under the SearchBlock. This field is used internally.
SUBOBJ	Integer	It refers to the address of other user interface components defined under the SearchBlock. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the SearchBlock would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the SearchBlock would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the SearchBlock would be automatically recalculated when the screen is resized.

2.2.6. **MenuBar**

MenuBar is a user interface component that contains Menu interface components. MenuBar can only be created within a Frame interface component.

The fields within the MenuBar interface component are explained below along with their intended use (***Table 2.7***).

Table 2.7 *MenuBar User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the MenuBar is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the MenuBar. This field is used internally.
NOOFSUB	Short	It refers to the number of Menu components defined under the MenuBar. This field is used internally.
SUBMENU	Integer	It refers to the address of Menu components defined under the MenuBar. This field is used internally.

2.2.7. **Menu**

A Menu is a user interface component that contains Menu and MenuItem interface components. Menus can only be created within a MenuBar or Menu interface components.

The fields within the menu interface component are explained below along with their intended use (***Table 2.8***).

Table 2.8 *Menu User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the MenuBar is kept.
TEXT	Char(30)	It refers to the field where the text of the Menu displayed on the screen is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Menu. This field is used internally.
NOOFSUB	Short	It refers to the number of Menu and MenuItem components defined under the Menu.
SUBOBJ	Integer	It refers to the address of the Menu and MenuItem elements defined under the Menu. This field is used internally.

2.2.8. **MenuItem**

MenuItem is a clickable user interface component. MenuItems can only be created within a Menu interface component.

The fields within the MenuItem interface component and their intended use are explained below (***Table 2.9***).

Table 2.9 *MenuItem User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the MenuItem is kept.
TEXT	Char(30)	It refers to the field where the text of the MenuItem displayed on the screen is kept.
SHORTCUT	Char(10)	It refers to the keyboard shortcut to activate the MenuItem.
HANDLE	Integer	It refers to the corresponding object in the program for the MenuItem. This field is used internally.
IMAGE	Integer	It refers to the image to be displayed for the MenuItem. This is an optional field.
ON-COMMAND	Integer	It refers to the function that will be called when the MenuItem is clicked.
ON-COMMAND-IND	Char(1)	It refers to the type of function that will be called when the MenuItem is clicked. This field is used internally.

2.2.9. **ToolBar**

A ToolBar is a user interface component that contains Tool interface components. ToolBar can only be created within a Frame interface component.

The fields within the ToolBar interface component and their intended use are explained below (***Table 2.10***).

Table 2.10 *ToolBar User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the ToolBar is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the ToolBar. This field is used internally.
TOP	Short	It refers to the top starting point of the ToolBar within the screen.
LEFT	Short	It refers to the left starting point of the ToolBar within the screen.
WIDTH	Short	It refers to the width of the ToolBar on the screen.
HEIGHT	Short	It refers to the height of the ToolBar on the screen.
NOOFSUB	Short	It refers to the number of Tool components defined under the ToolBar. This field is used internally.
SUBTOOL	Integer	It refers to the address of Tool components defined under the ToolBar. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the ToolBar would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the ToolBar would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the ToolBar would be automatically recalculated when the screen is resized.

2.2.10. **Tool**

Tool is a user interface component that can be clicked. Tool can only be created within a ToolBar interface component.

The fields within the Tool interface component are explained below along with their intended use (***Table 2.11***).

Table 2.11 *Tool User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the ToolBar is kept.
LABEL	Char(30)	It refers to the field where the text displayed when the tool is hovered, is kept.
HANDLE	Integer	It refers to the corresponding object for the tool in the program. This field is used internally.
TOOL_COUNTER	Integer	It refers to the field where the Tool's position within the Toolbar is kept. This field is used internally.
IMAGE	Integer	It refers to the image that is displayed for the Tool. This is an optional field.
ON-CLICK	Integer	It refers to the function that will be called when the Tool is clicked.
ON-CLICK-IND	Char(1)	It refers to the type of function that will be called when the tool is clicked. This field is used internally.

2.2.11. **Button**

Button is a clickable user interface component that does not contain any interface components. It can be created within interface components such as Frame, Block, or Tab that contain interface components.

The fields within the Button interface component are explained below along with their intended use (***Table 2.12***).

Table 2.12 *Button User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Button is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Button. This field is used internally.
TOP	Short	It refers to the top starting point of the Button within the screen.
LEFT	Short	It refers to the left starting point of the Button within the screen.
WIDTH	Short	It refers to the width of the Button on the screen.
HEIGHT	Short	It refers to the height of the Button on the screen.
ON-CLICK	Integer	It refers to the function that will be called when the Button is clicked.
ON-CLICK-IND	Char(1)	It refers to the type of function that will be called when the Button is clicked. This field is used internally.
IMAGE	Integer	It refers to the image that is displayed for the Button. This is an optional field.
REPOINT	Bool	It refers whether the top and left points of the Button would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the Button would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the Button would be automatically recalculated when the screen is resized.

2.2.12. **BitmapButton**

BitmapButton is a clickable interface component that doesn't contain any interface elements. A BitmapButton can be created within interface components such as Frame, Block, and Tab that contain interface components.

The fields within the BitmapButton interface component are explained below along with their intended use (***Table 2.13***).

Table 2.13 *BitmapButton User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Button is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Button. This field is used internally.
TOP	Short	It refers to the top starting point of the Button within the screen.
LEFT	Short	It refers to the left starting point of the Button within the screen.
WIDTH	Short	It refers to the width of the Button on the screen.
HEIGHT	Short	It refers to the height of the Button on the screen.
ON-CLICK	Integer	It refers to the function that will be called when the Button is clicked.
ON-CLICK-IND	Char(1)	It refers to the type of function that will be called when the Button is clicked. This field is used internally.
IMAGE	Integer	It refers to the image that is displayed for the Button.
REPOINT	Bool	It refers whether the top and left points of the Button would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the Button would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the Button would be automatically recalculated when the screen is resized.

2.2.13. **TextCtrl**

TextCtrl refers to a user interface component that does not contain any interface components and allows user to enter values. TextCtrl can be created within interface components such as Frame, Block, and Tab that contain other interface components.

The fields within the TextCtrl interface component and their intended use are explained below (***Table 2.14***).

Table 2.14 *TextCtrl User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the TextCtrl is kept.
TEXT	Char(30)	It refers to the field for text displayed left to the area where value of TextCtrl is entered.
HANDLE	Integer	It refers to the corresponding object in the program for the TextCtrl. This field is used internally.
TOP	Short	It refers to the top starting point of the TextCtrl within the screen.
LEFT	Short	It refers to the left starting point of the TextCtrl within the screen.
WIDTH	Short	It refers to the width of the TextCtrl on the screen.
HEIGHT	Short	It refers to the height of the TextCtrl on the screen.
TYPE	LongLong	It refers to the field where the reference type of TextCtrl is kept within the program.
TNAME	Char(30)	It refers to the name of the corresponding type of TextCtrl. This field is used internally.
TYPE_DEF	LongLong	It refers to the field where the reference type of TextCtrl's description field is kept.
TNAME_DEF	Char(30)	It refers to the name of the corresponding type of TextCtrl's description field. This field is used internally.
BINDS_TO	Integer	It refers to the field where the program object (internal structure) to which the TextCtrl is bound, is kept.

BINDS_TO_POS	Integer	It refers to the field where the column information of the program object (internal structure) to which the TextCtrl is bound, is kept. This field is used internally.
BINDS_TO_DEF	Integer	It refers to the field where the program object (internal structure) to which the TextCtrl's description field is bound, is kept.
BINDS_TO_DEF_POS	Integer	It refers to the field where the column information of the program object (internal structure) to which the TextCtrl's description field is bound, is kept. This field is used internally.
FRAME	Integer	It refers to the reference of the Frame interface component that would be displayed when the button that is automatically displayed when the TextCtrl is hovered over, is clicked.
ON-CHANGE	Integer	It refers to the function that would be called when the value of TextCtrl is changed.
ON-CHANGE-IND	Char(1)	It refers to the type of function that will be called when the value of TextCtrl is changed. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the TextCtrl would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the TextCtrl would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the TextCtrl would be automatically recalculated when the screen is resized.

2.2.14. **Grid**

Grid is a user interface component that contains column interface components and can display data in a tabular format. Grid can be created within interface components such as Frame, Block, and Tab that contain other interface components. The data contained within the grid interface component cannot be modified. Operations such as sorting and subtotaling can be performed on the data within the grid interface component.

The fields within the Grid interface component are explained below along with their intended use. (***Table 2.15***)

Table 2.15 *Grid User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Grid is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the Grid. This field is used internally.
TOP	Short	It refers to the top starting point of the Grid within the screen.
LEFT	Short	It refers to the left starting point of the Grid within the screen.
WIDTH	Short	It refers to the width of the Grid on the screen.
HEIGHT	Short	It refers to the height of the Grid on the screen.
BINDS_TO	Integer	It refers to the field where the program object (internal table) to which the Grid is bound, is kept.
BINDS_TO_POS	Integer	It refers to the field where the column information of the program object (internal table) to which the Grid is bound, is kept. This field is used internally.
ON-DOUBLE-CLICK	Integer	It refers to the function that will be called when Grid is double clicked.
ON-DOUBLE-CLICK-IND	Char(1)	It refers to type of the function that will be called when Grid is double clicked. This field is used internally.

NOOFCOLS	Short	It refers to the number of Column interface components defined under the Grid.
COLS	Integer	It refers to the address of the Column interface components defined under the Grid. This field is used internally.
SELROWS	Integer	It refers to the internal table reference that holds the selected rows for the grid. This field is used internally.
SUBLEVEL	Short	It refers to the field where the information about number of columns that the subtotaling is made for the grid, is kept. This field is used internally.
SORTLEVEL	Short	It refers to the field where the information about number of columns that the sorting is made for the grid, is kept. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the Grid would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the TextCtrl would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the TextCtrl would be automatically recalculated when the screen is resized.

2.2.15. **Column**

Column is a user interface component that represents a column of tabular data. Columns can only be created within the Grid and TableCtrl interface elements.

The fields within the Column interface element are explained below along with their intended use. (***Table 2.16***)

Table 2.16 *Column User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Column is kept.
TITLE	Char(30)	It refers to the column label displayed for the column.
LEFT	Short	It refers to the left starting point of the column within the top-level interface component.
WIDTH	Short	It refers to the width of the column on the screen.
BINDS_TO	Integer	It refers to the field where the internal table column information to which the column is bound, is kept.
FRAME	Integer	It refers to the field where the frame information that would be opened when the button that is automatically displayed when a cell to which the Column is bound is clicked, is kept.
SUBLEVEL	Short	If a column is used in a subtotal, this is the field where the information about the position of column within the subtotal, is kept.
SORTLEVEL	Short	If a column is used in a sort, this is the field where the information about the position of column within the sort, is kept.
ON-CHANGE	Integer	It refers to the function that would be called when a cell to which the column is bound, is changed.
ON-CHANGE-IND	Char(1)	It refers to the type of function that would be called when a cell to which the column is bound, is changed. This field is used internally.

2.2.16. **TableCtrl**

TableCtrl is an user interface component that contains Column interface components and can display data in a tabular format. TableCtrl's can be created within interface components such as Frame, Block, and Tab that contain interface components. Data contained within a TableCtrl interface component can be modified. Operations such as sorting or subtotaling can not be performed on data within a TableCtrl interface component.

The fields within the TableCtrl interface component, along with their intended use, are explained below. (***Table 2.17***)

Table 2.17 *TableCtrl User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the TableCtrl is kept.
HANDLE	Integer	It refers to the corresponding object in the program for the TableCtrl. This field is used internally.
TOP	Short	It refers to the top starting point of the TableCtrl within the screen.
LEFT	Short	It refers to the left starting point of the TableCtrl within the screen.
WIDTH	Short	It refers to the width of the TableCtrl on the screen.
HEIGHT	Short	It refers to the height of the TableCtrl on the screen.
BINDS_TO	Integer	It refers to the field where the program object (internal table) to which the TableCtrl is bound, is kept.
BINDS_TO_POS	Integer	It refers to the field where the column information of the program object (internal table) to which the TableCtrl is bound, is kept. This field is used internally.
ON-DOUBLE-CLICK	Integer	It refers to the function that will be called when TableCtrl is double clicked.
ON-DOUBLE-CLICK-IND	Char(1)	It refers to type of the function that will be called when TableCtrl is double clicked. This field is used

		internally.
NOOFCOLS	Short	It refers to the number of Column interface components defined under the TableCtrl.
COLS	Integer	It refers to the address of the Column interface components defined under the TableCtrl. This field is used internally.
SELROWS	Integer	It refers to the internal table reference that holds the selected rows for the TableCtrl. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the TableCtrl would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the TableCtrl would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the TableCtrl would be automatically recalculated when the screen is resized.

2.2.17. **SearchOption**

SearchOption is a user interface component used in reporting that contains a table of start and end values. SearchOption can be created within interface components such as Frame, Block, and Tab that contain other interface components.

If a change is made within the SearchOption interface component, the table bound to SearchOption, is automatically updated. There is no need to write any code for this update.

The fields within the SearchOption interface component, along with their intended use, are described below. (***Table 2.18***)

Table 2.18 *SearchOption User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the SearchOption is kept.
ID	Char(30)	It refers to the field that is searched for the searchoption when searchoption is used in the select operation. The ID field should be unique for all SearchOption definitions within the program, meaning that an ID value can only be used by one SearchOption.
TEXT	Char(30)	It refers to the field that to the text displayed to the left of the values of SearchOption.
HANDLE	Integer	It refers to the corresponding object in the program for the SearchOption. This field is used internally.
TOP	Short	It refers to the top starting point of the SearchOption within the screen.
LEFT	Short	It refers to the left starting point of the SearchOption within the screen.
WIDTH	Short	It refers to the width of the SearchOption on the screen.
HEIGHT	Short	It refers to the height of the SearchOption on the screen.
TYPE	LongLong	It refers to the field where the reference type of SearchOption is kept.
TNAME	Char(30)	It refers to the name of the corresponding type of TextCtrl.

		This field is used internally.
FRAME	Integer	It refers to the reference of the Frame interface component that would be displayed when the button that is automatically displayed when the low and high values of SearchOption are hovered over, is clicked.
FRAME_RANGE	Integer	It refers to the reference of the Frame interface component that would be displayed when the button that is displayed right to the values of SearchOption, is clicked.
REPOINT	Bool	It refers whether the top and left points of the SearchOption would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the SearchOption would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the SearchOption would be automatically recalculated when the screen is resized.

2.2.18. **DropDown**

DropDown is a user interface component that does not contain any interface elements and allows you to select a value from a list. DropDowns can be created within interface components such as Frame, Block, and Tab that contain other interface components.

The fields within the DropDown interface component and their intended use are explained below (***Table 2.19***).

Table 2.19 *DropDown User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the DropDown is kept.
TEXT	Char(30)	It refers to the field that to the text displayed to the left of the values of DropDown.
HANDLE	Integer	It refers to the corresponding object in the program for the DropDown. This field is used internally.
TOP	Short	It refers to the top starting point of the DropDown within the screen.
LEFT	Short	It refers to the left starting point of the DropDown within the screen.
WIDTH	Short	It refers to the width of the DropDown on the screen.
HEIGHT	Short	It refers to the height of the DropDown on the screen.
BINDS_TO	Integer	It refers to the field where the program object (internal table) to which the DropDown is bound, is kept.
BINDS_TO_POS	Integer	It refers to the field where the column information of the program object (internal table) to which the DropDown is bound, is kept. This field is used internally.
ON-CHANGE	Integer	It refers to the function that would be called when the value of DropDown is changed.

ON-CHANGE-IND	Char(1)	It refers to the type of function that will be called when the value of DropDown is changed. This field is used internally.
REPOINT	Bool	It refers whether the top and left points of the DropDown would be automatically recalculated when the screen is resized.
RESIZE-W	Bool	It refers whether the width of the DropDown would be automatically recalculated when the screen is resized.
RESIZE-H	Bool	It refers whether the height of the DropDown would be automatically recalculated when the screen is resized.

2.2.19. **Image**

Image is a user interface component that displays a picture object and does not contain any interface components. Image interface component currently can not be created within any interface component. An image interface component can be referenced by interface components such as Button, BitmapButton, MenuItem, and Tool.

The fields within the Image interface component are explained below along with their intended use (***Table 2.20***).

Table 2.20 *Image User Interface Component*

Field Name	**Data Type**	**Purpose of use**
NAME	Char(30)	It refers to the field where the name of the Image is kept.
SRC	Char(30)	It refers to the source file location that is relative to the program location for the Image.
HANDLE	Integer	It refers to the corresponding object in the program for the Image. This field is used internally.

2.3. VERD Language Basic Constructs

2.3.1. Internal Table

Internal tables are objects that consist of multiple rows of a type defined within a program. Internal tables consist of multiple rows of the same type.

Internal tables are defined within the program using the **data** keyword. All database tables in the database specified with "**database**" component can be used directly to define internal tables. Examples of internal table definitions are given below.

```
data p_mustab type table of mus.
data p_mustab like table of mus.
```

Code Sample 2.1 *Internal Table Definition Samples*

2.3.2. Internal Structure

Internal structures are objects that consist of a single row of a type defined within the program.

Internal structures are defined within the program using the "**data"** keyword. All database tables in the database specified with "**database**" component can be used directly to define internal structures. Examples of internal structures definitions are given below.

```
data p_mus type mus.
data p_mus like mus.
```

Code Sample 2.2 *Internal Structure Definition Samples*

2.3.3. **Parameter**

Parameters are objects used for the basic built-in types. Parameters for the following basic built-in types can be defined for the program:

- Char
- Unsigned Char
- Short
- Unsigned Short
- Int
- Unsigned Int
- Long
- Unsigned Long
- Float
- Double
- String
- Date
- Time
- DateTime

Parameters are defined within the program using the **data** keyword. Examples of parameter definitions are given below:

```
data p_int type int.
data p_string type string.
data p_float type float
data p_date type date.
```

Code Sample 2.3 *Parameter Definition Samples*

2.3.4. **Internal Structure Reference**

Internal structure references are dynamically created objects consisting of a single row of a type defined within the program. Automatic reference counting is implemented for internal structure references, and if all strong references pointing to an internal structure reference are cleared from memory, the memory occupied by the internal structure reference would be automatically cleared. The most important difference between internal structure references and internal structures is that they are dynamically created, and clearing the memory occupied by them is subject to certain conditions (e.g. a strong reference count of 0).

Internal structure references might be **weak** or **strong**. Weak references just borrows reference from an another reference. Weak references do not affect memory clearing process. In other words, memory occupied by reference is not cleared if the number of weak references drops to 0. In addition it is not possible to create a new reference object by using weak references. Strong references conversely affect memory clearing process. In other words, memory occupied by reference is cleared if the number of strong references drops to 0. In addition it is possible to create a new reference object by using strong references. References are by default strong. If it is necessary to define a weak reference, it should be declared explicitly.

Internal structure references are defined within the program using the "**data"** keyword. All database tables in the database specified with "**database**" component can be used directly to define internal structure references. Examples of internal structure reference definitions are given below.

```
data p_mus type ref to mus .
data p_mus type weak ref to mus .
```

Code Sample 2.4 *Internal Structure Reference Definition Samples*

2.3.5. **Internal Table References**

Internal table references are dynamically created objects that refer a table of a type defined within the program. Automatic reference counting is implemented for internal table references, and if all strong references pointing to an internal table reference are cleared from memory, the memory occupied by the internal table reference would be automatically cleared. The most significant difference between internal table references and internal tables is that they are dynamically created, and clearing the memory occupied by them is subject to certain conditions (e.g., a strong reference count of 0).

Internal table references might be **weak** or **strong**. Weak references just borrows reference from an another table reference. Weak table references do not affect memory clearing process. In other words, memory occupied by table reference is not cleared if the number of weak references drops to 0. In addition it is not possible to create a new table reference object by using weak references. Strong table references conversely affect memory clearing process. In other words, memory occupied by table reference is cleared if the number of strong table references drops to 0. In addition it is possible to create a new table reference table by using strong table references. References are strong by default. If it is necessary to define a weak table reference, it should be declared explicitly.

Internal table references are defined within the program using the "**data"** keyword. All database tables in the database specified with "**database**" component can be used directly to define internal table references. Examples of internal table reference definitions are given below.

```
data p_mus_tab type ref to table of mus .
data p_mus_tab type weak ref to table of mus .
```

Code Sample 2.5 *Internal Table Reference Definition Samples*

2.3.6. **Built-in SY Internal Structure**

VERD language has a built-in internal structure **SY** (short for system**)** whose type is built-in **SYS** type. VERD language has **SY** internal structure since it is required to check the results of operations such as SELECT, INSERT, UPDATE, DELETE, READ TABLE, etc. **SY** internal structure has also been used for other purposes since it keeps fields such as user name, language and universal unique identifier (UUID) for program.

The fields within the **SY** internal structure (**SYS** type) are explained below along with their intended use (***Table 2.21***).

Table 2.21 *Fields of built-in SYS Type*

Field Name	**Data Type**	**Purpose of use**
RESULT	Integer	It refers to the field where the result of operations such as SELECT, INSERT, READ TABLE is kept.
LANGU	Char(2)	It refers to field where the language of program is kept. It can be used to display texts in the language that user logged in.
USRNAME	Char(30)	It refers to field where the user name of program is kept.
DATE	Date	It refers to field where the running date of program is kept.
TIME	Time	It refers to field where the running time of program is kept.
UUID	Char(36)	It refers to field where the universal unique identifier for the program is kept.

2.3.6.1. **SY internal structure usage after SELECT operation**

```
data lc_sklt type sklt .

data lc_message_type  type string .
data p_message type string .

select single * from sklt  into corresponding
                           fields of lc_sklt
                           where skltid = p_lock_entry
                              and uuid ne sy-uuid.

 if sy-result = 1 .

   lc_message_type = "W" .
   concatenate p_message ,  " is locked by user " , lc_sklt-klid into p_message.
   message p_message type lc_message_type .

 endif.
```

Code Sample 2.6 *SY internal structure usage Sample 1*

In the example above (***Code Sample 2.6***), the record in the **sklt** table whose **skltid** is the value in **p_lock_entry** and whose **uuid** is not the value in **sy-uuid** is transferred to the lc_sklt internal structure with SELECT SINGLE operation. If there is such record the value of **sy-result** is set to 1 by runtime else value of **sy-result** is set to 0. The result of operation is checked after SELECT SINGLE operation and if result is 1 then message is displayed to the user by using MESSAGE construct.

2.3.6.2. **SY internal structure usage after INSERT operation**

```
data lc_sklt type sklt .

data lc_message_type  type string .
data p_message type string .

insert into sklt from lc_sklt .

 if sy-result = 0 .

   lc_message_type = "W" .
   concatenate p_message ,  " is locked by user " , lc_sklt-klid into p_message.
   message p_message type lc_message_type .

 endif.
```

Code Sample 2.7 *SY internal structure usage Sample 2*

In the example above (***Code Sample 2.7***), the record indicated by **lc_sklt** internal structure is inserted into table **sklt** table that is defined in database. If insertion operation is success then the value of **sy-result** is set to 1 by runtime else value of **sy-result** is set to 0. The result of operation is checked after INSERT operation and if result is 0 then message is displayed to the user by using MESSAGE construct.

2.3.6.3. **SY internal structure usage after UPDATE operation**

```
data lc_ksbsy type ksbsy .

data lc_message_type  type string .
data p_message type string .

update ksbsy from lc_ksbsy.

if sy-result = 0 .

  lc_message_type = "W" .
  concatenate  "Table KSBSY is not updated for “, lc_ksbsy-syid into p_message.
  message p_message type lc_message_type .

 endif.
```

Code Sample 2.8 *SY internal structure usage Sample 3*

In the example above (***Code Sample 2.8***), the record indicated by **lc_ksbsy** internal structure is updated in table **ksbsy** table that is defined in database. If update operation is success then the value of **sy-result** is set to 1 by runtime else value of **sy-result** is set to 0. The result of operation is checked after UPDATE operation and if result is 0 then message is displayed to the user by using MESSAGE construct.

2.3.6.4. **SY internal structure usage after DELETE operation**

```
data lc_ksbsy type ksbsy .

data lc_message_type  type string .
data p_message type string .

delete ksbsy from lc_ksbsy.

if sy-result = 0 .

  lc_message_type = "W" .
  concatenate  "Table KSBSY is not deleted for ", lc_ksbsy-syid into p_message.
  message p_message type lc_message_type .

 endif.
```

Code Sample 2.9 *SY internal structure usage Sample 4*

In the example above (***Code Sample 2.9***), the record indicated by **lc_ksbsy** internal structure is deleted from table **ksbsy** table that is defined in database. If delete operation is success then the value of **sy-result** is set to 1 by runtime else value of **sy-result** is set to 0. The result of operation is checked after DELETE operation and if result is 0 then message is displayed to the user by using MESSAGE construct.

2.3.6.5. **SY internal structure usage after READ TABLE operation**

```
data lc_uobrd type uobrd .
data lc_uobrd_ref type weak ref to uobrd .

read table p_uobrd_tab into lc_uobrd_ref
                        where obid = p_sblk->sobid
                          and dlid = sy-langu .
if sy-result = 0 .

  select single * from uobrd into corresponding
                              fields of lc_uobrd
                              where obid = p_sblk->sobid
                                and dlid = sy-langu .
 endif.
```

Code Sample 2.10 *SY internal structure usage Sample 5*

In the example above (***Code Sample 2.10***), the first record from the **p_uobrd_tab** table whose **obid** is the value in **sobid** field of **p_sblk** internal structure field and whose **dlid** is the value in **langu** field of built-in internal structure **sy**, is referenced by the **lc_uobrd_ref** nternal structure reference. If there is such record in **p_uobrd_tab** internal table**,** then the value of **sy-result** is set to 1 by runtime else value of **sy-result** is set to 0.

The result of operation is checked after READ TABLE operation and if result is 0 then it is checked whether there is such record in database table **uobrd** by using SELECT SINGLE construct.

2.3.7. **Assignment Operation**

Assignment operation in VERD language is similar to assignment operation in other languages. The operands are separated with **equal** (=) operator and value of right operand is assigned to left operand. Assignment operation with **equal** (=) operator is only valid in stand-alone assignments, in IF-ELSE and WHILE constructs, **equal** (=) operator behaves as comparison operator instead of assignment operator. Left and right operand in assignment operation might be internal structure fields, internal structure reference fields and parameters.

For numeric assignments if both operands are not in same type then right operand is cast to left operand type. For character assignmens, if left operand is fixed length character field and length of right operand is bigger than length of left operand, then right operand is truncated to fit in length of left operand. If length of left operand is dynamic (built-in **string** type) and length of right operand is bigger than length of left operand, then left operand is extended to fit in length of right operand. Examples for assignment operation are given below.

2.3.7.1. **Assignment operation for numeric fields**

```
type compi >>   ksbkid type fksb-ksbkid,
                klmkt type fksb-kstm.

data p_compi type compi.

data p_kstm type double.
data p_ksdg type double.

data p_int type int .
data p_short type short .

p_kstm =  p_compi-klmkt.
p_ksdg = p_kstm .

p_int = p_ksdg .
p_short = p_int .
```

Code Sample 2.11 *Assignment Operation Sample 1*

In the example above (***Code Sample 2.11***), **p_compi** internal structure whose type is program-defined **compi** type, **p_kstm** and **p_ksdg** paremeters whose type are built-in **double**

type **p_int** parameter whose type is built-in **int** type and p_short parameter whose type is built-in **short** type, are defined.

In first assignment, value in **p_compi-klmkt** internal structure field is assigned to **p_kstm** parameter. Since **p_compi-klmkt** has type **fksb-kstm** and **kstm** field of **fksb** table in database is in **double** type**,** no casting is performed in this assignment.

In second assignment, value in **p_kstm** parameter is assigned to **p_ksdg** parameter. Since **p_kstm** and **p_ksdg** parameters both is in **double** type**,** no casting is performed in this assignment.

In third assignment, value in **p_ksdg** parameter is assigned to **p_int** parameter. Since **p_kstm** parameter is in **double** type and **p_int** parameter is in **int** type **,** casting from **double to int** is performed in this assignment.

Finally in fourth assignment, value in **p_int** parameter is assigned to **p_short** parameter. Since **p_int** parameter is in **int** type and **p_short** parameter is in **short** type**,** casting from **int to short** is performed in this assignment.

2.3.7.2. **Assignment operation for character fields where left operand has fixed length**

```
data lc_sklt type sklt .
data p_lock_entry type string .

 lc_sklt-skltid = p_lock_entry .
 lc_sklt-klid =sy-usrname .
```

Code Sample 2.12 *Assignment Operation Sample 2*

In the example above (***Code Sample 2.12***), **lc_sklt** internal structure whose type is **sklt** table defined in database type and **p_lock_entry** paremeter whose type is built-in **string** type, are defined. **SY** internal structure is built-in internal structure defined by compiler and fields of **SY** internal structure such as **result** and **uuid** is automatically changed by runtime.

In first assignment, value in **p_lock_entry** parameter is assigned to **lc_sklt-skltid** internal structure field. Since **p_lock_entry** has built-in **string** type and **skltid** field of **sklt** table in database is fixed length character field with 40 chars**,** if length of **p_lock_entry** is greater than 40 chars then only first 40 chars of **p_lock_entry** would be assigned to **lc_sklt-skltid**. If length of **p_lock_entry** is lower than 40 chars then all chars of **p_lock_entry** would be assigned to **lc_sklt-skltid** and remaining chars of **lc_sklt-skltid** field would be assigned to zero.

In second assignment, value in **sy-usrname** internal structure field is assigned to **lc_sklt-klid** internal structure field. Since **usrname** field of **sys** built-in type is fixed length character field with 30 chars and **klid** field of **sklt** table in database is fixed length character field with 20 chars only first 20 chars of **sy-usrname** would be assigned to **lc_sklt- klid**.

2.3.7.3. **Assignment operation for character fields where left operand has dynamic length**

```
data lc_sklt type sklt .
data p_skltid type string .

 p_skltid = lc_sklt-skltid.
```

Code Sample 2.13 *Assignment Operation Sample 3*

In the example above (***Code Sample 2.13***), **lc_sklt** internal structure whose type is **sklt** table defined in database type and **p_skltid** paremeter whose type is built-in **string** type, are defined.

As it can be seen in above example, value in to **lc_sklt-skltid** internal structure field is assigned to **p_lock_entry** parameter. Since **p_lock_entry** has built-in **string** type and **skltid** field of **sklt** table in database is fixed length character field with 40 chars**,** if length of **p_lock_entry** is greater than 40 chars then 40 chars of **lc_sklt-skltid** would be assigned to **p_lock_entry** and length of **p_lock_entry** would be updated as 40**.** If length of **p_lock_entry** is lower than 40 chars then firstly **p_lock_entry** would be extended to include 40 chars after that 40 chars of **lc_sklt-skltid** would be assigned to **p_lock_entry.**

2.3.8. **Arithmetic Operations**

Arithmetic operations in VERD language are similar to arithmetic operations in other languages. Parantheses can be used during arithmetic operations in VERD language. Only numeric field types such as **short, int, float, double** etc. can be used in arithmetic operations. Currently only **plus** (+), **subtract** (-), **multiply** (*) and **divide** (/) operations are supported in arithmetic operations. Precedence for operators and parantheses are same with other languages. Parantheses has the the highest precende followed by **multiply** and **divide** then **plus** and **subtract**. Examples for arithmetic operations are given below.

2.3.8.1. **Arithmetic operations without parantheses**

```
data p_compi type compi.

data p_kstm type double.
data p_ksdg type double.

data p_kstpdgr type double .

p_kstpdgr =  p_compi-klmkt * p_ksdg + p_kstm.
```

Code Sample 2.14 *Arithmetic Operations Sample 4*

In the example above (***Code Sample 2.14***), **p_compi** internal structure whose type is program-defined **compi** type, **p_kstm**, **p_ksdg**, and **p_kstpdgr** paremeters whose type are built-in **double** type, are defined. Firstly arithmetic operation for multiplication **p_compi-klmkt * p_ksdg** is calculated. Then **p_kstm** is added to calculated value and the result is assigned to **p_kstpdgr** parameter.

2.3.8.2. **Arithmetic operations with parantheses**

```
data p_compi type compi.

data p_kstm type double.
data p_ksdg type double.

data p_kstpdgr type double .

data lc_hkts type double .
data lc_kkts type double .

p_kstpdgr =  p_compi-klmkt * p_ksdg * lc_kkts / ( lc_hkts * p_kstm ) .
```

Code Sample 2.15 *Arithmetic Operations Sample 2*

In the example above (***Code Sample 2.15***), **p_compi** internal structure whose type is program-defined **compi** type, **p_kstm**, **p_ksdg**, **p_kstpdgr**, **lc_hkts** and **lc_hkts** paremeters whose type are built-in **double** type, are defined. Firstly arithmetic operation in paranthesis (lc_hkts * p_kstm) is calculated . Then calculation of **p_compi-klmkt * p_ksdg * lc_kkts** is performed. Finally second calculated value is divided by first calculated and the result is assigned to **p_kstpdgr** parameter .

2.3.9. **TYPE Language Construct**

The TYPE construct is a language construct that allows for defining types within a program. Types defined with TYPE construct can be used while defining program variables by using DATA construct. Every program variable (internal structure, parameters, internal structure references, internal tables and internal table references) must have a type either built-in, database table or defined by using TYPE construct.

Within TYPE construct, fields of the type are defined. Each field of a type must have a type either built-in, database table or defined by using TYPE construct. "**Include**" keyword can be used to define a field within TYPE construct. In this case, all fields of the type specified with "**Include**" keyword are added to the type that is defined with TYPE construct.

Within TYPE construct, a field of the type can be internal structure reference. To define an internal structure reference within TYPE construct, "**type ref to**" or "**type weak ref to**" statements can be used. In former case the internal structure reference for the field within TYPE construct would be **strong** reference, in latter case it would be **weak** reference. In other words, internal structure references within TYPE construct are **strong** by default, to make it **weak**, it should have to explicitly stated.

Similarly within TYPE construct, a field of the type can be internal table reference. To define an internal table reference within TYPE construct, "**type ref to table of**" or "**type weak ref to table of**" statements can be used. In former case the internal table reference for the field within TYPE construct would be **strong** reference, in latter case it would be **weak** reference. In other words, internal table references within TYPE construct are **strong** by default, to make it **weak**, it should have to explicitly stated.

TYPE construct has two forms. In first form, TYPE construct begins with "**begin of type**" statement. In this form, TYPE construct should have to be terminated by using "**end of type**" or "**endtype**" statements. In second form, TYPE construct begins with "**type**" keyword followed by type name and after type name ">>" follows. In this form, TYPE construct terminates with **dot** (**"."**).

2.3.9.1. **TYPE Construct usage with "begin of type" keyword and without "include" keyword**

```
begin of type sthk_ca .

    shbid type sthk-shbid ,
    shkid type sthk-shkid,
    blgt type sthb-blgt ,
    mzism type vmzgd-mzism ,
    istism type uistd-istism ,
    uyism type uuyrd-uyism.

endof type .
```

Code Sample 2.16 *Type Construct Sample 1*

In the example above (***Code Sample 2.16***), **sthk_ca** type is defined with "**begin of type**" statement and its definition is terminated with "**endof type"** statement. Within **sthk_ca** type, **shbid** field whose type is **sthk-shbid, shkid** field whose type is **sthk-shkid, blgt** field whose type is **sthb-blgt, mzism** field whose type is **vmzgd-mzism, istism** field whose type is **uistd-istism** and **uyism** field whose type is **uuyrd-uyism** are defined.

All fields defined within **sthk_ca** type are separated with **comma**("**,**"). After defining **sthk_ca** type it can be used with "**data**" keyword to define porgram variables.

2.3.9.2. **TYPE Construct usage with "begin of type" and "include" keywords**

```
begin of type sthk_ca .

    include type sthk,
    blgt type sthb-blgt ,
    mzism type vmzgd-mzism ,
    istism type uistd-istism ,
    uyism type uuyrd-uyism.

endof type .
```

Code Sample 2.17 *Type Construct Sample 2*

In the example above (***Code Sample 2.17***), **sthk_ca** type is defined with "**begin of type**" statement and its definition is terminated with "**endof type"** statement. Within **sthk_ca** type,
all fields in database table **sthk** are added to **sthk_ca** type by using "**include**"keyword. Additionally, **blgt** field whose type is **sthb-blgt, mzism** field whose type is **vmzgd-mzism, istism** field whose type is **uistd-istism** and **uyism** field whose type is **uuyrd-uyism** are defined within **sthk_ca** type .

All fields defined within **sthk_ca** type are separated with **comma**(**",**"). After defining **sthk_ca** type it can be used with "**data**" keyword to define porgram variables.

2.3.9.3. **TYPE Construct usage with ">>" keyword and without "include" keyword**

```
type sthk_ca >> shbid type sthk-shbid ,
                shkid type sthk-shkid,
                blgt type sthb-blgt ,
                mzism type vmzgd-mzism ,
                istism type uistd-istism ,
                uyism type uuyrd-uyism.
```

Code Sample 2.18 *Type Construct Sample 3*

In the example above (***Code Sample 2.18***)), **sthk_ca** type is defined with "**type""** keyword followed by type name and ">>" keyword and its definition is terminated with **dot**(**".**") . Within **sthk_ca** type, **shbid** field whose type is **sthk-shbid, shkid** field whose type is **sthk-shkid, blgt** field whose type is **sthb-blgt, mzism** field whose type is **vmzgd-mzism, istism** field whose type is **uistd-istism** and **uyism** field whose type is **uuyrd-uyism** are defined.

All fields defined within **sthk_ca** type are separated with **comma**(**",**"). After defining **sthk_ca** type it can be used with "**data**" keyword to define porgram variables.

2.3.9.4. **TYPE Construct usage with “>>” and “include” keywords**

```
type sthk_ca >>  include type sthk,
                 blgt type sthb-blgt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.
```

Code Sample 2.19 *Type Construct Sample 4*

In the example above (***Code Sample 2.19***), **sthk_ca** type is defined with “**type”**” keyword followed by type name and “>>” keyword and its definition is terminated with **dot**(“**.**”). Within **sthk_ca** type, all fields in database table **sthk** are added to **sthk_ca** type by using “**include**”keyword. Additionally, **blgt** field whose type is **sthb-blgt, mzism** field whose type is **vmzgd-mzism, istism** field whose type is **uistd-istism** and **uyism** field whose type is **uuyrd-uyism** are defined within **sthk_ca** type .

All fields defined within **sthk_ca** type are separated with **comma**(“**,**”). After defining **sthk_ca** type it can be used with “**data**” keyword to define porgram variables.

2.3.10. **CONSTANT Language Construct**

The CONSTANT construct is a language construct that allows to define constants that can be used in other language constructs such as SELECT, LOOP, WHILE, IF-ELSE etc. With CONSTANT construct, numeric constants, string constant and date time constants can be defined. Examples for CONSTANT construct are given below.

2.3.10.1. **CONSTANT Construct usage for numeric values**

```
constant zero value 0.
constant max_unsigned_short value 65535 .

constant pi_value value 3.14
```

Code Sample 2.20 *Constant Construct Sample 1*

In the example above (***Code Sample 2.20***), the numeric constant **zero** which has value 0, the numeric constant **max_unsigned_short** which has value 65535 and the float numeric constant **pi_value** which has value 3.14, are defined.

2.3.10.2. **CONSTANT Construct usage for string values**

```
constant lock_tab_stock_mov value “STHB”.
constant sign_percent value "%"
```

Code Sample 2.21 *Constant Construct Sample 2*

In the example above (***Code Sample 2.21***), the string constant **lock_tab_stock_mov** which has value **STHB and** the string constant sign_percent which has value **%** is defined. String constants should have to placed within two double quotes for compiler to recognize them as string constants.

2.3.10.3. **CONSTANT Construct usage for date time values**

```
constant zero_date value "00.00.4712 BC" .
constant zero_time value "00:00:00" .
constant zero_datetime value "00.00.4712 BC 00:00:00" .

constant date_beg value “23.12.2025”
constant time_beg value “18:00:00”
constant datetime_beg value “23.12.2025 18:00:00”
```

Code Sample 2.22 *Constant Construct Sample 3*

In the example above (***Code Sample 2.22***), the date time constants **zero_date, zero_time**, **zero_datetime, date_beg, time_beg** and **datetime_beg** are defined**.**

As it can be see from examples above, date and time constants should have to placed within two double quotes. Addititionaly date values should have to be in format “DD.MM.YYYY”, time values should have to be in format “HH:MM:SS” and datetime value should have to be in format “DD.MM.YYYY HH:MM:SS” for compiler to recognize them as date and time constants.

For date values before BC, “**BC**” or “**BCE**” value should have to be added after year with space sepearated, in other words for date values before BC, date values should have to be in format “DD.MM.YYYY **BC**” or “DD.MM.YYYY **BCE**”. Similarly for datetime values before BC datetime values should have to be in format “DD.MM.YYYY **BC** HH:MM:SS” or “DD.MM.YYYY **BCE** HH:MM:SS”.

2.3.11. **NEW Language Construct**

The NEW construct is a language construct that allows allocating memory for internal structure references. Internal structure references do not refer anything when they are defined. There are two options to make internal structure references to refer data. In first option, they can refer data if they are assigned from another internal structure reference that is already referring data. In second option, they can refer data if they are assigned to data allocated by NEW construct.

Only strong internal structure references can be assigned by using NEW construct. Weak internal structure references can only refer data by using only first option, by being assigned from another internal structure reference that is already referring data. Example for NEW construct is given below.

2.3.11.1. **NEW construct usage with parameter**

```
data lc_vmzv_ref type ref to vmzv.

lc_vmzv_ref = new vmzv .
```

Code Sample 2.23 *New Construct Sample 1*

In the example above (***Code Sample 2.23***), the strong **lc_vmzv_ref** internal structure reference, whose type is **vmzv** table defined in the database, is defined. Using the NEW construct, the **lc_vmzv_ref** internal structure reference is assigned to data allocated by VERD runtime.

2.3.11.1. **NEW construct usage with internal structure field**

```
type vmzv_ca >>  include type vmzv,
                 vmzd_ref type ref to vmzd .

data lc_vmzv type vmzv_ca.

 lc_vmzv- vmzd_ref = new vmzd .
```

Code Sample 2.24 *New Construct Sample 2*

In the example above (***Code Sample 2.24***), **vmzv_ca** type and **lc_vmzv** internal structure, whose type is **vmzv_ca** are defined**.** The **vmzv_ca** type includes all fields of **vmzv** database table and **vmzd_ref** field that is a strong reference to **vmzd** table defined in the database .

Using the NEW construct, the **vmzd_ref** field of **lc_vmzv** internal structure is assigned to data allocated by VERD runtime.

2.3.12. **NEW TABLE OF Language Construct**

The NEW TABLE OF construct is a language construct that allows allocating an internal table for internal table references. Internal table references do not refer anything when they are defined. There are two options to make internal table references to refer an internal table. In first option, they can refer an internal table if they are assigned from another internal table reference that is already referring an internal table. In second option, they can refer an internal table if they are assigned to an internal table allocated by NEW TABLE construct.

Only strong internal table references can be assigned by using NEW TABLE OF construct. Weak internal table references can only refer an internal table by using only first option, by being assigned from another internal table reference that is already referring an internal table. Example for NEW TABLE OF construct is given below.

2.3.12.1. **NEW TABLE OF construct usage with parameter**

```
data lc_vmzv_tab_ref type ref to table of vmzv.

lc_vmzv_tab_ref = new table vmzv .
```

Code Sample 2.25 *New Table Of Construct Sample 1*

In the example above (***Code Sample 2.25***), the **lc_vmzv_tab_ref** internal table reference, whose type is **vmzv** table defined in the database, is defined. Using the NEW TABLE OF construct, the **lc_vmzv_tab_ref** internal table reference is assigned to an internal table allocated by VERD runtime.

2.3.12.2. **NEW TABLE OF construct usage with internal structure field**

```
type ubnk_ca >>   include  type ubnk  ,
                  ubnks_tab type ref to table of ubnks .

data lc_ubnk type ubnk_ca.

 lc_ubnk-ubnks_tab = new table of ubnk_ca .
```

Code Sample 2.26 *New Table Of Construct Sample 2*

In the example above (***Code Sample 2.26***), **ubnk_ca** type and **lc_ubnk** internal structure, whose type is **ubnk_ca** are defined**.** The **ubnk_ca** type includes all fields of **ubnk** database table and **ubnks_tab** field that is a strong reference to a internal table of **ubnks** table defined in the database .

Using the NEW TABLE OF construct, the **ubnks_tab** field of **lc_ubnk** internal structure is assigned to data allocated by VERD runtime.

2.3.13. **ROUTINE Language Construct**

The ROUTINE construct is a language construct that allows defining functions that can be called with EXEC language struct. Routines are very similar to functions in other programming languages. Routines can be called with routine parameters. Evey routine has a signature that defines the required parameters to call a routine.

Routines can include other language constructs such as SELECT, LOOP, WHILE, IF-ELSE etc. Routine parameters are divided into three subgroups. Imported parameters of a routine (with "**importing**" keyword) are parameters that are read but not changed by routine. Exported parameters of a routine (with "**exporting**" keyword) are parameters that are not read but changed by routine. Finally, modified parameters of a routine (with "**modifying**" keyword) are parameters that are both read and changed by routine.

Every routine parameter has a type defined with "**type**" keyword. Types of routine parameters can be any type of variable that can be defined with DATA construct including internal structure, internal structure references, parameters, internal tables and internal table references. Internal structure fields can not be used while defining routine parameters.

While calling a routine with EXEC construct, compiler automatically checks whether type of variable in EXEC struct matches with type of routine parameter. Additionaly, compiler checks whether the subgroup of variable (**import, export, modify**) matches with subgorup of the routine parameter.

Subgroups of routine parameters are intended for just easy readability of routine. There is currently no check for subgroups. For example, imported parameters are not checked whther they are changed or not within routine. In the future, checks for subgroups migt be implemented.

ROUTINE construct should have to be closed with "**endroutine**" keyword that indicates termination of routine body. Examples for ROUTINE construct are given below.

2.3.13.1. **ROUTINE Construct usage with no parameters**

```
routine on_change_istid .

   data lc_uist like uist .
   data lc_sthk type weak ref to sthk_ca.

   data lc_message type string .
   data lc_message_type  type string .

   select single * from  uist  into corresponding
                               fields of  lc_uist
                               where istid = gd_sthbk-istid .
   if sy-result = 0 .

      lc_message = "Movement type is not defined" .
      lc_message_type = "E" .
      message lc_message type lc_message_type .

   endif.

   loop at gd_sthk_tab into lc_sthk .

      if  lc_sthk->istid ne gd_sthbk-istid  .

         lc_sthk->istid = gd_sthbk-istid .
         if lc_sthk->updind = updind_dbexists.
            lc_sthk->updind = updind_update.
         endif.

      endif.

   endloop .

   refresh frame sthr_ch_details .

endroutine .
```

Code Sample 2.27 *Routine Construct Sample 1*

In the example above (***Code Sample 2.27***), routine **on_change_istid** that has no parameters, is defined. This type of routines that takes no parameters, are generally defined as event-handling routines that are executed as a response to user actions since event-handling routines should be called with no parameters.

Within routine, **lc_uist** internal structure, **lc_sthk** weak internal structure reference, **lc_message** and **lc_message_type** parameters are defined. As it can be seen from above example, various language constructs such as SELECT SINGLE, IF-ELSE, LOOP, REFRESH FRAME are used. The routine was terminated with "**endroutine**" statement.

2.3.13.2. **ROUTINE Construct usage with imported, exported and modified parameters**

```
routine det_conv_factor importing p_source_unit type string
                                  p_target_unit type string
                         modifying p_uobd_tab type table of uobd
                         exporting  p_hkts type double
                                    p_kkts type double .

  data lc_uobd type uobd .
  data lc_uobd_ref type weak ref to uobd .

  read table p_uobd_tab into lc_uobd_ref
                          where hobid = p_target_unit
                            and kobid = p_source_unit .


  if sy-result = 0 .

      select single * from uobd into corresponding
                                  fields of lc_uobd
                                  where hobid = p_target_unit
                                    and kobid = p_source_unit .

      if sy-result =  1 .

        append lc_uobd to p_uobd_tab .
        p_hkts = lc_uobd-hkts .
        p_kkts = lc_uobd-kkts .

      endif.

  else .

     p_hkts = lc_uobd_ref->hkts .
     p_kkts = lc_uobd_ref->kkts .

  endif.

endroutine .
```

Code Sample 2.28 *Routine Construct Sample 2*

In the example above (***Code Sample 2.28***), routine **det_conv_factor** that takes five parameters, is defined. Two of parameters(**p_source_unit**, **p_target_unit**) are imported, one of them(**p_uobd_tab**) is modified and two of them(**p_hkts**, **p_kkts**) are exported. Imported parameters(**p_source_unit**, **p_target_unit)** are only read, exports parameters are only changed(**p_hkts**, **p_kkts**) and modified paramater is both read and changed(**p_uobd_tab**).

Within routine, **lc_uobd** internal structure and **lc_uobd_ref** weak internal structure reference, are defined. As it can be seen from above example, various language constructs such as READ, SELECT, IF-ELSE, APPEND are used. The routine was terminated with "**endroutine**" statement.

2.3.14. **EXEC Language Construct**

The EXEC construct is a language construct that allows calling routines. Routines can be called with EXEC construct with routine parameters.

While calling a routine with EXEC construct, compiler automatically checks whether type of variable in EXEC struct matches with type of routine parameter. Additionaly, compiler checks whether the subgroup of variable (**import, export, modify**) matches with subgorup of the routine parameter.

Imported routine parameters specified with "**importing**" keyword should be specified with "**exporting**" keyword while calling the routine with EXEC struct. Similarly exported routine parameters specified with "**exporting**" keyword should be specified with "**importing**" keyword while calling the routine with EXEC struct. Modified routine parameters specified with "**modified**" keyword should be specified without any change while calling the routine with EXEC struct. Examples for EXEC construct are given below.

2.3.14.1. **EXEC Construct usage with no parameters**

```
routine on_change_istid .


endroutine .
exec on_change_istid
```

Code Sample 2.29 *Exec Construct Sample 1*

In the example above (***Code Sample 2.29***), routine **on_change_istid** that has no parameters, is defined. The **on_change_istid** routine is called with EXEC struct. Since **on_change_istid** routine has no input parametersi, the routine is called with EXEC struct without specifying any parameter.

2.3.14.2. **EXEC Construct usage with imported, exported and modified parameters**

```
routine det_conv_factor importing p_source_unit type string
                                  p_target_unit type string
                        modifying p_uobd_tab type table of uobd
                        exporting  p_hkts type double
                                  p_kkts type double .



endroutine .

data lc_source_unit type string .
data lc_target_unit type string .

data g_ uobd_tab type type table of uobd.

data lc_hkts type double .
data lc_kkts type double .

exec det_conv_factor exporting lc_source_unit
                               lc_target_unit
                     modifying g_uobd_tab
                     importing  lc_hkts
                                lc_kkts.
```

Code Sample 2.30 *Exec Construct Sample 2*

In the example above (***Code Sample 2.30***), routine **det_conv_factor** that takes five parameters, is defined. Two of parameters(**p_source_unit**, **p_target_unit**) are imported, one of them(**p_uobd_tab**) is modified and two of them(**p_hkts**, **p_kkts**) are exported. Imported parameters(**p_source_unit**, **p_target_unit)** are only read, exports parameters are only changed(**p_hkts**, **p_kkts**) and modified paramater is both read and changed(**p_uobd_tab**).

To call the **det_conv_factor** routine with EXEC struct, **lc_source_unit**, **lc_target_unit** parameters whose type are built-in **string** type, **g_ uobd_tab tab** internal table whose type is uobd table defined in database and **lc_hkts, lc_kkts** parameters whose type are built-in **double** type, are defined. Routine is called with EXEC struct by specifying **imported**, **exported** and **modified** parameters. As it can be seen from example, **imported** parameters of routine are called with "**exporting**" keyword and **exported** parameters of routine are called with "**importing**" keyword.

2.3.15. **SELECT Language Construct**

The SELECT construct is a language construct that allows for database querying and is very similar to the Select construct in SQL. Program variables (internal structure, parameters, internal structure references, internal tables and internal table references and search-options) can be used within the SELECT construct, making working with the database much easier.

The SELECT construct transfers the data it retrieves from the database to the internal table defined within the SELECT construct. The internal table to be used within the SELECT construct is specified with the "**into corresponding fields of table**" statement.

In case where the * keyword is used within the SELECT construct the compiler automatically checks for similarities between the data types of the internal table and the database table. If the data types are the same, it transfers all columns from the database table to the internal table. If the data types are not the same, but the internal table data type includes the database table data type (using the **include** construct), it also transfers all columns from the database table to the corresponding columns within the internal table. If the data types are not the same and the internal table does not include the database table data type, the compiler determines the common fields between the two data types, and the SQL query sent to the database is modified to include only these fields. As a result of the query, only the fields of the internal table that are common with database table are transferred to the internal table.

If * is not used in the SELECT construct, only the columns specified in the SELECT construct in the database table are transferred to the internal table.

2.3.15.1. **SELECT Construct usage without “where” keyword**

```
data gs_usbld_tab type table of usbld .

select * from usbld into
        corresponding fields of table
        gs_usbld_tab.
```

Code Sample 2.31 *Select Construct Sample 1*

In the example above (***Code Sample 2.31***), the internal table **gs_usbld_tab**, which has a type of **usbld** that is a database table, is defined. With the SELECT construct, all records in the **usbld** table are transferred to the **gs_usbld_tab** internal table.

2.3.15.2. **SELECT Construct usage with “where” keyword and internal structure**

```
data g_vmzg type vmzg .
data lc_umzg_tab  type table of umzg .

select * from  umzg
               into corresponding
               fields of table lc_umzg_tab
               where mzgid = g_vmzg-mzgid .
```

Code Sample 2.32 *Select Construct Sample 2*

In the example above (***Code Sample 2.32***), the internal structure **g_vmzg**, whose type is the **vmzg** table defined in the database, and the internal table **lc_umzg_tab**, whose type is the **umzg** table defined in the database, are defined. With the SELECT construct, all records in the **umzg** table whose **mzgid** is the value in **g_vmzg-mzgid** are transferred to the **lc_umzg_tab** table.

2.3.15.3. **SELECT Construct usage with "where" keyword and parameter**

```
data lc_umzg_tab  type table of umzg .
data p_string type string .

select * from  umzg
               into corresponding
               fields of table lc_umzg_tab
               where mzgid = p_string .
```

Code Sample 2.33 *Select Construct Sample 3*

In the example above (***Code Sample 2.33***), the internal table **lc_umzg_tab**, whose type is the **umzg** table defined in the database, and the parameter **p_string**, whose type is the built-in type **string**, are defined. With the SELECT construct, all records in the **umzg** table whose **mzgid** is the value in **p_string** are transferred to the **lc_umzg_tab** table.

2.3.15.4. **SELECT Construct usage with "where" and "and" keyword**

```
data g_vmzg type vmzg .
data p_string type string .
data gs_usbld_tab type table of usbld .

 select * from usbld into
          corresponding fields of table
          gs_usbld_tab.
          where sblid = g_vmzg-sblid
              and dlid = p_string .
```

Code Sample 2.34 *Select Construct Sample 4*

In the example above (***Code Sample 2.34***), the internal structure **g_vmzg,** whose type is the **vmzg** table defined in the database, the parameter **p_string,** whose type is the built-in type **string**, and the internal table **gs_usbld_tab**, whose type is the **usbld** table defined in the database, are defined. With the SELECT construct, all records in the **usbld** table whose **sblid** is the value in **g_vmzg-sblid** and whose **dlid** is the value in **p_string** are transferred to the **gs_usbld_tab** table.

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, all records in the **usbld** table whose **sblid** is the value in **g_vmzg-sblid** or whose **dlid** is the value in **p_string** are transferred to the **gs_usbld_tab** table.

2.3.15.5. **SELECT Construct usage with declaring column names and without "where" keyword**

```
data gs_usbld_tab type table of usbld .

select sblid,dlid from usbld into
        corresponding fields of table
        gs_usbld_tab.
```

Code Sample 2.35 *Select Construct Sample 5*

In the example above (***Code Sample 2.35***), the internal table **gs_usbld_tab**, which is a **usbld** table defined in the database, are defined. With the SELECT construct, only the **sblid** and **dlid** columns from all records in the **usbld** table were transferred to the **gs_usbld_tab table**; data in other columns was not retrieved from the database.

2.3.15.6. **SELECT Construct usage with declaring column names, "where" keyword and internal structure**

```
data g_vmzg type vmzg .
data lc_vmzg_tab type table of vmzg .

select mzid,ensid,mztid from  vmzg
                into corresponding
                fields of table lc_vmzg_tab
                where mzid = g_vmzg-mzid .
```

Code Sample 2.36 *Select Construct Sample 6*

In the example above (***Code Sample 2.36***), the internal structure **g_vmzg**, whose type is a **vmzg** table defined in the database, and the internal table **lc_vmzg_tab**, whose type is also a **vmzg** table, are defined. With the SELECT construct, only the **mzid**, **ensid**, and **mztid** columns from all records in the **vmzg** table whose **mzid** is the value in **g_vmzg-mzid** are transferred to the **lc_vmzg_tab** table.

2.3.15.7. **SELECT Construct usage with declaring column names, "where" keyword and parameter**

```
data p_string type string .
data lc_vmzg_tab type table of vmzg .

select mzid,ensid,mztid from  vmzg
                    into corresponding
                    fields of table lc_vmzg_tab
                    where mzid = p_string.
```

Code Sample 2.37 *Select Construct Sample 7*

In the example above (***Code Sample 2.37***), the **p_string** parameter, whose type is a built-in **string** type, and the **lc_vmzg_tab** internal table, whose type is the **vmzg** table defined in the database, are defined. Using the SELECT construct, only the **mzid**, **ensid**, and **mztid** columns from all records in the **vmzg** table whose **mzid** is the value in the **p_string** are transferred to the **lc_vmzg_tab** table.

2.3.15.8. **SELECT Construct usage with declaring column names, using "where" and "and" keywords**

```
data g_vmzg type vmzg .
data p_string type string .
data lc_vmzg_tab type table of vmzg .

select mzid,ensid,mztid from  vmzg
                    into corresponding
                    fields of table lc_vmzg_tab
                    where mzid = g_vmzg-mzid
                       and ensid = p_string .
```

Code Sample 2.38 *Select Construct Sample 8*

In the example above (***Code Sample 2.38***), the **g_vmzg** internal structure, whose type **vmzg** table defined in the database, the **p_string** parameter, whose type is a built-in **string** type, and the **lc_vmzg_tab** internal table, which is has also **vmzg** table type, are defined. Using the select construct, only the **mzid**, **ensid**, and **mztid** columns from all records in the **vmzg** table, where the **mzid** is the value in **g_vmzg-mzid** and the **ensid** is the value in **p_string** are transferred to the **lc_vmzg_tab** table.

In the example above, **“or”** keyword can also be used instead of **“and”** keyword. In this case, only the **mzid**, **ensid**, and **mztid** columns from all records in the **vmzg** table, where the **mzid** is the value in **g_vmzg-mzid** or the **ensid** is the value in **p_string** are transferred to the **lc_vmzg_tab** table.

2.3.16. **SELECT SINGLE Language Construct**

The SELECT SINGLE construct is quite similar to the SELECT construct. The only difference between the SELECT SINGLE construct and the SELECT construct is that the SELECT SINGLE construct transfers the data retrieved from the database to an internal structure defined within the SELECT SINGLE construct. While the object transferred in the SELECT construct is an internal table, in the SELECT SINGLE construct, this object is an internal structure. In other words, while the SELECT construct retrieves multiple records from the database, the SELECT SINGLE construct retrieves only a single record. The internal structure to be used within the SELECT SINGLE construct is specified with the expression "**into corresponding fields of**".

All the examples shown for the SELECT construct can be converted into the SELECT SINGLE construct by using the "**single**" keyword immediately after the "**Select**" keyword, by changing the "**into corresponding fields of table**" statements to "**into corresponding fields of**", and by using an internal structure instead of an internal table. Examples for SELECT SINGLE constructs are given below:

2.3.16.1. **SELECT SINGLE Construct usage without "where" keyword**

```
data gs_usbld type usbld .

select single * from usbld into
        corresponding fields of
        gs_usbld.
```

Code Sample 2.39 *Select Single Construct Sample 1*

In the example above (***Code Sample 2.39***), the internal structure **gs_usbld**, whose type is usbld table defined in the database, is defined. The first record from the **usbld** table is transferred into the **gs_usbld** structure using the SELECT SINGLE construct.

2.3.16.2. **SELECT SINGLE Construct usage with "where" keyword and internal structure**

```
data g_vmzg type vmzg .
data lc_umzg  type umzg .

select single * from  umzg
                into corresponding
                fields of lc_umzg
                where mzgid = g_vmzg-mzgid .
```

Code Sample 2.40 *Select Single Construct Sample 2*

In the example above (***Code Sample 2.40***), the **g_vmzg** internal structure, whose type vmzg table defined in the database, and the **lc_umzg** internal structure, whose type is a **umzg** table defined in the database, are defined. Using the SELECT SINGLE construct, the first record in the **umzg** table whose **mzgid** is the value in **g_vmzg-mzgid** is transferred into the lc_**umzg** structure.

2.3.16.3. **SELECT SINGLE Construct usage with "where" keyword and parameter**

```
data lc_umzg  type table of umzg .
data p_string type string .

select single * from  umzg
                into corresponding
                fields of lc_umzg
                where mzgid = p_string .
```

Code Sample 2.41 *Select Single Construct Sample 3*

In the example above (***Code Sample 2.41***), the internal structure **lc_umzg**, whose type is the **umzg** table defined in the database, and the **p_string** parameter, whose type is an built-in type **string**, are defined. Using the SELECT SINGLE construct, the first record in the **umzg** table whose **mzgid** is the value in **p_string** is transferred into the **lc_umzg** structure.

2.3.16.4. **SELECT SINGLE Construct usage with "where" and "and" keyword**

```
data g_vmzg type vmzg .
data p_string type string .
data gs_usbld type usbld .

 select single * from usbld into
          corresponding fields of
          gs_usbld.
          where sblid = g_vmzg-sblid
             and dlid = p_string .
```

Code Sample 2.42 *Select Single Construct Sample 4*

In the example above (***Code Sample 2.42***), the **g_vmzg** internal structure, whose type is the **vmzg** table defined in the database, the **p_string** parameter, whose type is the built-in type **string**, and the **gc_usbld** internal structure, whose type is the **usbld** table defined in the database, are defined. Using the SELECT SINGLE structure, the first record from the **usbld** table, whose **sblid** is the value in **g_vmzg-sblid** and whose **dlid** is the value in **p_string**, is transferred into the **gs_usbld** structure.

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, the first record from the **usbld** table, whose **sblid** is the value in **g_vmzg-sblid** or whose **dlid** is the value in **p_string**, is transferred into the **gs_usbld** structure.

2.3.16.5. **SELECT SINGLE Construct usage with declaring column names and without "where" keyword**

```
data gs_usbld type usbld .

select single sblid,dlid from usbld into
                        corresponding fields of gs_usbld.
```

Code Sample 2.43 *Select Single Construct Sample 5*

In the example above (***Code Sample 2.43***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, is defined. Using the SELECT SINGLE construct, only the **sblid** and **dlid** columns from the first record in the **usbld** table are transferred into the **gs_usbld** structure; the data in the other columns is not retrieved from the database.

2.3.16.6. **SELECT SINGLE Construct usage with declaring column names, using "where" keyword and internal structure**

```
data g_vmzg type vmzg .
data lc_vmzg type vmzg .

select single mzid,ensid,mztid from  vmzg
                                into corresponding
                                fields of lc_vmzg
                                where mzid = g_vmzg-mzid .
```

Code Sample 2.44 *Select Single Construct Sample 6*

In the example above (***Code Sample 2.44***), the **g_vmzg** internal structure, whose type is **vmzg** table defined in the database, and the **lc_vmzg** internal structure, whose type is also **vmzg** table, are defined. Using the SELECT SINGLE structure, only the **mzid**, **ensid**, and **mztid** columns from the first record in the **vmzg** table whose **mzid** is the value in **g_vmzg-mzid** are transferred to the **lc_vmzg** structure.

2.3.16.7. **SELECT SINGLE Construct usage with declaring column names, using "where" keyword and parameter**

```
data p_string type string .
data lc_vmzg type vmzg .

select single mzid,ensid,mztid from  vmzg
                into corresponding
                fields of lc_vmzg
                where mzid = p_string.
```

Code Sample 2.45 *Select Single Construct Sample 7*

In the example above (***Code Sample 2.45***), the **p_string** parameter, whose type is built-in **string** type, and the **lc_vmzg** internal structure, whose type is the **vmzg** table defined in the database, are defined. Using the SELECT SINGLE structure, only the **mzid**, **ensid**, and **mztid** columns from the first record in the **vmzg** table whose **mzid** is the value in the **p_string** are transferred into the **lc_vmzg** structure.

2.3.16.8. **SELECT SINGLE Construct usage with declaring column names, using "where" and "and" keyword**

```
data g_vmzg type vmzg .
data p_string type string .
data lc_vmzg type vmzg .

select mzid,ensid,mztid from  vmzg
                  into corresponding
                  fields of lc_vmzg
                  where mzid = g_vmzg-mzid
                     and ensid = p_string .
```

Code Sample 2.46 *Select Single Construct Sample 8*

In the example above (***Code Sample 2.46***), the **g_vmzg** internal structure, whose type is the **vmzg** table defined in the database, the **p_string** parameter, whose type is the built-in **string** type, and the **lc_vmzg** internal structure, whose type is also the **vmzg** table, are defined. Using the SELECT SINGLE construct, only the **mzid**, **ensid**, and **mztid** columns from the first record in the **vmzg** table, whose **mzid** is the value in **g_vmzg-mzid** and whose **ensid** is the value in **p_string**, are transferred to the **lc_vmzg** structure.

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, only the **mzid**, **ensid**, and **mztid** columns from the first record in the **vmzg** table, whose **mzid** is the value in **g_vmzg-mzid** or whose **ensid** is the value in **p_string**, are transferred to the **lc_vmzg** structure.

2.3.17. **SELECT JOIN Language Construct**

The SELECT JOIN construct is a language construct that allows for database querying and is very similar to the Select Join construct in SQL. Program variables (internal structure, parameters, internal structure references, internal tables and internal table references and search-options) can be used within the SELECT JOIN construct, making working with the database much easier.

The main difference between SELECT JOIN and SELECT construct is in SELECT JOIN construct, the data can be retrieved from multiple database tables. Database tables that data would be received should be joined by using “**inner join**” ,”**left join**” or “**right join**” statements. Similar to Select Join construct in SQL, conditions that database tables would be joined can be specified after “**inner join**” ,”**left join**” or “**right join**” statements. Also database tables can be abbreviated by using “**as**” keyword.

The SELECT JOIN construct transfers the data it retrieves from the database to the internal table defined within the SELECT JOIN construct. The internal table to be used within the SELECT JOIN construct is specified with the "**into corresponding fields of table**" statement.

In case where the * keyword is used within the SELECT JOIN construct the compiler automatically checks for similarities between the data types of the internal table and the database tables used in SELECT JOIN construct. The compiler determines the common fields between the type of internal table and database tables used in SELECT JOIN construct, and the SQL query sent to the database is modified to include only these fields. As a result of the query, only the fields of the internal table that are common with database table are transferred to the internal table.

If * is not used in the SELECT JOIN construct, only the columns specified in the SELECT JOIN construct in the database table are transferred to the internal table.

2.3.17.1. **SELECT JOIN Construct usage without "where" keyword**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 blyr type unsigned int ,
                 blmt type unsigned short ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data gd_sthk_tab type table of sthk_ca.

select * from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
        left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab .
```

Code Sample 2.47 *Select Join Construct Sample 1*

In the example above (***Code Sample 2.47***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, are defined. With the SELECT JOIN construct, all records in the **sthk** table are joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **gd_sthk_tab** internal table. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

2.3.17.2. **SELECT JOIN Construct usage with "where" keyword and internal structure**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data gd_sthk_tab type table of sthk_ca.
data g_sthk type sthk_ca.

select * from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
        left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = g_sthk-shbid .
```

Code Sample 2.48 *Select Join Construct Sample 2*

In the example above (***Code Sample 2.48***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, and **g_sthk** internal structure whose type is also **sthk_ca** are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **g_sthk-shbid**, are joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **gd_sthk_tab** internal table. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

2.3.17.3. **SELECT JOIN Construct usage with "where" keyword and parameter**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data gd_sthk_tab type table of sthk_ca.
data lc_shbid type string.

select * from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
        left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = lc_shbid .
```

Code Sample 2.49 *Select Join Construct Sample 3*

In the example above (***Code Sample 2.49***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, and **lc_shbid** parameter whose type is built-in **string** type, are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **lc_shbid**, are joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **gd_sthk_tab** internal table. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

2.3.17.4. **SELECT JOIN Construct usage with "where" and "and" keyword**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data gd_sthk_tab type table of sthk_ca.
data lc_shbid type string.
data lc_mzid type string.

select * from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
        left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = lc_shbid
           and a-mzid = lc_mzid .
```

Code Sample 2.50 *Select Join Construct Sample 4*

In the example above (***Code Sample 2.50***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, **lc_shbid** and **lc_mzid** parameters whose type are built-in **string** type, are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **lc_shbid** and **mzid** is the value in **lc_mzid**, are joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **gd_sthk_tab** internal table. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, records in the **sthk** table whose **shbid** is the value in **lc_shbid** or **mzid** is the value in **lc_mzid**, are joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **gd_sthk_tab** internal table.

2.3.17.5. **SELECT JOIN Construct usage with declaring column names and without "where" keyword**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data gd_sthk_tab type table of sthk_ca.

select a-mzid,b-blgt,c-mzism from  sthk as a
       inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
       left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
       into corresponding fields of table gd_sthk_tab
```

Code Sample 2.51 *Select Join Construct Sample 5*

In the example above (***Code Sample 2.51***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, are defined. With the SELECT JOIN construct, all records in the **sthk** table are joined with tables **sthb** and **vmzgd**, then transferred to the **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **gd_sthk_tab** internal table.

2.3.17.6. **SELECT JOIN Construct usage with declaring column names, "where" keyword and internal structure**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data gd_sthk_tab type table of sthk_ca.
data g_sthk type sthk_ca.

select a-mzid,b-blgt,c-mzism from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = g_sthk-shbid .
```

Code Sample 2.52 *Select Join Construct Sample 6*

In the example above (***Code Sample 2.52***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, and **g_sthk** internal structure whose type is also **sthk_ca,** are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **g_sthk-shbid** are joined with tables **sthb** and **vmzgd**, then transferred to the **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **gd_sthk_tab** internal table.

2.3.17.7. **SELECT JOIN Construct usage with declaring column names, “where keyword and parameter**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data gd_sthk_tab type table of sthk_ca.
data lc_shbid type lc_shbid.

select a-mzid,b-blgt,c-mzism from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = lc_shbid.
```

Code Sample 2.53 *Select Join Construct Sample 7*

In the example above (***Code Sample 2.53***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, and **lc_shbid** parameter whose type is built-in **string** type**,** are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **lc_shbid** are joined with tables **sthb** and **vmzgd**, then transferred to the **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **“a”**, table **sthb** is abbreviated as **“b”**, and table **vmzgd** is abbreviated as **“c” .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **“a”**), **blgt** from table **sthb**(abbreviated as **“b”**), **mzism** from table **vmzgd** (abbreviated as **“c”**) are transferred from database into the **gd_sthk_tab** internal table.

.

2.3.17.8. **SELECT JOIN Construct usage with declaring column names, using "where" and "and" keywords**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data gd_sthk_tab type table of sthk_ca.
data lc_shbid type lc_shbid.
data lc_mzid type lc_mzid.

select a-mzid,b-blgt,c-mzism from  sthk as a
        inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
        left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
        into corresponding fields of table gd_sthk_tab
        where a-shbid = lc_shbid
           and a-mzid = lc_mzid.
```

Code Sample 2.54 *Select Join Construct Sample 8*

In the example above (***Code Sample 2.54***), **sthk_ca** type and **gd_sthk_tab** internal table whose type is **sthk_ca**, , **lc_shbid** and **lc_mzid** parameters whose type are built-in **string** type**,** are defined. With the SELECT JOIN construct, records in the **sthk** table whose **shbid** is the value in **lc_shbid** and whose **mzid** is the value in **lc_mzid** are joined with tables **sthb** and **vmzgd**, then transferred to the **gd_sthk_tab** internal table.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **gd_sthk_tab** internal table.

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, records in the **sthk** table whose **shbid** is the value in **lc_shbid** or **mzid** is the value in **lc_mzid**, are joined with tables **sthb** and **vmzgd**, then transferred to the **gd_sthk_tab** internal table.

2.3.18. **SELECT SINGLE JOIN Language Construct**

The SELECT SINGLE JOIN construct is quite similar to the SELECT JOIN construct. The only difference between the SELECT SINGLE JOIN construct and the SELECT JOIN construct is that the SELECT SINGLE JOIN construct transfers the data retrieved from the database to an internal structure defined within the SELECT SINGLE JOIN construct. While the object transferred in the SELECT JOIN construct is an internal table, in the SELECT SINGLE JOIN construct, this object is an internal structure. In other words, while the SELECT JOIN construct retrieves multiple records from the database, the SELECT SINGLE JOIN construct retrieves only a single record. The internal structure to be used within the SELECT SINGLE JOIN construct is specified with the expression "**into corresponding fields of**".

All the examples shown for the SELECT JOIN construct can be converted into the SELECT SINGLE JOIN construct by using the "**single**" keyword immediately after the "**Select**" keyword, and by changing the "**into corresponding fields of table**" statements to "**into corresponding fields of**", and by using an internal structure instead of an internal table. Examples for SELECT SINGLE JOIN constructs are given below:

2.3.18.1. **SELECT SINGLE JOIN Construct usage without “where” keyword**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data g_sthk type sthk_ca.

select single * from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
              left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
              into corresponding fields of g_sthk.
```

Code Sample 2.55 *Select Single Join Construct Sample 1*

In the example above (***Code Sample 2.55***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table is joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **g_sthk** internal structure. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **“a”**, table **sthb** is abbreviated as **“b”**, table **vmzgd** is abbreviated as **“c”**, table **uistd** is abbreviated as **“d”** and table **uuyrd** is abbreviated as **“e” .**

2.3.18.2. **SELECT SINGLE JOIN Construct usage with "where" keyword and internal structure**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data g_sthk type sthk_ca.
data lc_sthb type sthb.

select single * from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
              left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_sthb-shbid .
```

Code Sample 2.56 *Select Join Construct Sample 2*

In the example above (***Code Sample 2.56***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, and **lc_sthb** internal structure whose type is **stbh** table defined in the database**,** are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_sthb-shbid**, is joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **g_sthk** internal structure. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

2.3.18.3. **SELECT SINGLE JOIN Construct usage with "where" keyword and parameter**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data g_sthk type table of sthk_ca.
data lc_shbid type string.

select single * from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
              left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_shbid.
```

Code Sample 2.57 *Select Join Construct Sample 3*

In the example above (***Code Sample 2.57***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, and **lc_shbid** parameter whose type is built-in **string** type, are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_shbid**, is joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **g_sthk** internal structure. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, table **vmzgd** is abbreviated as **"c"**, table **uistd** is abbreviated as **"d"** and table **uuyrd** is abbreviated as **"e" .**

2.3.18.4. **SELECT SINGLE JOIN Construct usage with “where” and “and” keywords**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 mzism type vmzgd-mzism ,
                 istism type uistd-istism ,
                 uyism type uuyrd-uyism.

data g_sthk type sthk_ca.
data lc_shbid type string.
data lc_mzid type string.

select single * from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              left join uistd as d on a-istid = d-istid and d-dlid = sy-langu
              left join uuyrd as e on a-uyid = e-uyid and e-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_shbid
                 and a-mzid = lc_mzid .
```

Code Sample 2.58 *Select Single Join Construct Sample 4*

In the example above (***Code Sample 2.58***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, **lc_shbid** and **lc_mzid** parameters whose type are built-in **string** type, are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_shbid** and **mzid** is the value in **lc_mzid**, is joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **g_sthk** internal structure. Since * keyword is used, compiler automatically determines which fields of each database table would be selected and then transferred to **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **“a”**, table **sthb** is abbreviated as **“b”**, table **vmzgd** is abbreviated as **“c”**, table **uistd** is abbreviated as **“d”** and table **uuyrd** is abbreviated as **“e” .**

In the example above, **“or”** keyword can also be used instead of **“and”** keyword. In this case, first record in the **sthk** table whose **shbid** is the value in **lc_shbid** or **mzid** is the value in **lc_mzid**, is joined with tables **sthb**, **vmzgd**, **uistd** and **uuyrd** then transferred to the **g_sthk** internal structure.

2.3.18.5. **SELECT SINGLE JOIN Construct usage with declaring column names and without "where" keyword**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data g_sthk type sthk_ca.

select single a-mzid,b-blgt,c-mzism from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              into corresponding fields of g_sthk .
```

Code Sample 2.59 *Select Single Join Construct Sample 5*

In the example above (***Code Sample 2.59***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table is joined with tables **sthb** and **vmzgd**, then transferred to the **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **g_sthk** internal structure.

2.3.18.6. **SELECT SINGLE JOIN Construct usage with declaring column names, "where" keyword and internal structure**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data g_sthk type sthk_ca.
data lc_sthb type sthb.

select single a-mzid,b-blgt,c-mzism from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_sthb-shbid.
```

Code Sample 2.60 *Select Single Join Construct Sample 6*

In the example above (***Code Sample 2.60***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, and **lc_sthb** internal structure whose type is **sthb** table defined in database**,** are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_sthb-shbid** is joined with tables **sthb** and **vmzgd**, then transferred to the **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **g_sthk** internal structure.

2.3.18.7. **SELECT SINGLE JOIN Construct usage with declaring column names, "where keyword and parameter**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data g_sthk type table of sthk_ca.
data lc_shbid type lc_shbid.

select single a-mzid,b-blgt,c-mzism from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_shbid.
```

Code Sample 2.61 *Select Join Construct Sample 7*

In the example above (***Code Sample 2.61***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, and **lc_shbid** parameter whose type is built-in **string** type**,** are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_shbid** is joined with tables **sthb** and **vmzgd**, then transferred to the **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **g_sthk** internal structure.

.

2.3.18.8. **SELECT SINGLE JOIN Construct usage with declaring column names, using "where" and "and" keywords**

```
type sthk_ca >>  include  type sthk ,
                 blgt type sthb-blgt ,
                 blyt type sthb-blyt ,
                 hrkt type sthb-hrkt ,
                 mzism type vmzgd-mzism .

data g_sthk type table of sthk_ca.
data lc_shbid type lc_shbid.
data lc_mzid type lc_mzid.

select single a-mzid,b-blgt,c-mzism from  sthk as a
              inner join sthb as b on a-shbid = b-shbid  and b-shby = a-shby
              left join vmzgd as c on a-mzid = c-mzid  and c-dlid = sy-langu
              into corresponding fields of g_sthk
              where a-shbid = lc_shbid
                 and a-mzid = lc_mzid.
```

Code Sample 2.62 *Select Construct Sample 8*

In the example above (***Code Sample 2.62***), **sthk_ca** type and **g_sthk** internal structure whose type is **sthk_ca**, , **lc_shbid** and **lc_mzid** parameters whose type are built-in **string** type**,** are defined. With the SELECT SINGLE JOIN construct, first record in the **sthk** table whose **shbid** is the value in **lc_shbid** and whose **mzid** is the value in **lc_mzid** is joined with tables **sthb** and **vmzgd**, then transferred to the **g_sthk** internal structure.

As it can be seen from the example above, table **sthk** is abbreviated as **"a"**, table **sthb** is abbreviated as **"b"**, and table **vmzgd** is abbreviated as **"c" .**

Since column names declared in the example, only **mzid** from table **sthk**(abbreviated as **"a"**), **blgt** from table **sthb**(abbreviated as **"b"**), **mzism** from table **vmzgd** (abbreviated as **"c"**) are transferred from database into the **g_sthk** internal structure.

In the example above, **"or"** keyword can also be used instead of **"and"** keyword. In this case, first record in the **sthk** table whose **shbid** is the value in **lc_shbid** or **mzid** is the value in **lc_mzid**, is joined with tables **sthb** and **vmzgd**, then transferred to the **g_sthk** internal structure.

2.3.19. **INSERT Language Construct**

The INSERT language construct allows a record to be inserted into a database table by using internal structure or internal structure references. To insert multiple records into a database table, internal tables or internal table references might also be used. In latter case, **"table"** keyword must be added after **"from"** keyword.

In the INSERT construct for single record insertion, type of internal structure or internal structure reference must include all key fields of database table. Similarly, in the INSERT construct for multiple records insertion, type of internal table or internal table reference must include all key fields of database table. Examples for INSERT construct are given below:

2.3.19.1. **INSERT Construct usage with internal structure**

```
data g_vmzv type vmzv .

insert vmzv from g_vmzv.
```

Code Sample 2.63 *Insert Construct Sample 1*

In the example above (***Code Sample 2.63***), the **g_vmzv** internal structure, whose type is **vmzv** table defined in the database, is defined. Using the INSERT construct, the record indicated by g_vmzv internal structure, would be inserted into **vmzv** database table.

2.3.19.2. **INSERT Construct usage with internal structure reference**

```
data g_vmzv_ref type ref to vmzv .

insert vmzv from g_vmzv_ref .
```

Code Sample 2.64 *Insert Construct Sample 2*

In the example above (***Code Sample 2.64***), the **g_vmzv_ref** internal structure reference, whose type is **vmzv** table defined in the database, is defined. Using the INSERT construct, the record indicated by data referenced by **g_vmzv_ref** internal structure reference, would be inserted into **vmzv** database table.

2.3.19.3. **INSERT Construct usage with internal table**

```
data g_vmzv_tab type table of vmzv .

insert vmzv from table g_vmzv_tab.
```

Code Sample 2.65 *Insert Construct Sample 3*

In the example above (***Code Sample 2.65***), the **g_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, is defined. Using the INSERT construct, records indicated by **g_vmzv_tab** internal table, would be inserted into **vmzv** database table.

2.3.19.4. **INSERT Construct usage with internal table reference**

```
data g_vmzv_ref_tab type ref to table of vmzv .

insert vmzv from table g_vmzv_ref_tab.
```

Code Sample 2.66 *Insert Construct Sample 4*

In the example above (***Code Sample 2.66***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, is defined. Using the INSERT construct, records indicated by the table referenced by **g_vmzv_ref_tab** internal table reference, would be inserted into **vmzv** database table.

2.3.20. **UPDATE Language Construct**

The UPDATE language construct allows a record in a database table to be updated by using internal structure or internal structure references. To update multiple records from a database table, internal tables or internal table references might also be used. In latter case, **"table"** keyword must be added after **"from"** keyword.

In the UPDATE construct for single record update, type of internal structure or internal structure reference must include all key fields of database table. Similarly, in the UPDATE construct for multiple records update, type of internal table or internal table reference must include all key fields of database table. Examples for UPDATE construct are given below:

2.3.20.1. **UPDATE Construct usage with internal structure**

```
data g_vmzv type vmzv .

update vmzv from g_vmzv.
```

Code Sample 2.67 *Update Construct Sample 1*

In the example above (***Code Sample 2.67***), the **g_vmzv** internal structure, whose type is **vmzv** table defined in the database, is defined. Using the UPDATE construct, the record in **vmzv** database table indicated by g_vmzv internal structure, would be updated in database.

2.3.20.2. **UPDATE Construct usage with internal structure reference**

```
data g_vmzv_ref type ref to vmzv .

update vmzv from g_vmzv_ref .
```

Code Sample 2.68 *Update Construct Sample 2*

In the example above (***Code Sample 2.68***), the **g_vmzv_ref** internal structure reference, whose type is **vmzv** table defined in the database, is defined. Using the UPDATE construct, the record in **vmzv** database table indicated by data referenced by **g_vmzv_ref** internal structure reference, would be updated in database.

2.3.20.3. **UPDATE Construct usage with internal table**

```
data g_vmzv_tab type table of vmzv .

update vmzv from table g_vmzv_tab.
```

Code Sample 2.69 *Update Construct Sample 3*

In the example above (***Code Sample 2.69***), the **g_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, is defined. Using the UPDATE construct, records in **vmzv** database table indicated by records in **g_vmzv_tab** internal table, would be updated in database.

2.3.20.4. **UPDATE Construct usage with internal table reference**

```
data g_vmzv_ref_tab type ref to table of vmzv .

update vmzv from table g_vmzv_ref_tab.
```

Code Sample 2.70 *Update Construct Sample 4*

In the example above (***Code Sample 2.70***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, is defined. Using the UPDATE construct, records in **vmzv** database table indicated by records in table referenced by **g_vmzv_ref_tab** internal table reference, would be updated in database.

2.3.21. **COMMIT Language Construct**

The COMMIT language construct allows all database related operations such as INSERT, UPDATE and DELETE to be committed to database. If database is specified with **autocommit** property false then without COMMIT construct, all INSERT, UPDATE and DELETE operations would be lost and not committed to database. Example for COMMIT construct is given below:

2.3.21.1. **COMMIT Construct usage**

```
data g_vmzv_tab type table of vmzv .
data g_vmzv_ref type ref to vmzv .

insert vmzv from g_vmzv_ref .
update vmzv from table g_vmzv_tab.

commit .
```

Code Sample 2.71 *COMMIT Construct Sample 1*

In the example above (***Code Sample 2.71***), the **gs_vmzv_tab** internal table and **g_vmzv_ref** internal structure reference, whose types are **vmzv** table defined in the database, are defined. INSERT and UPDATE operations for table **vmzv** are performed by using **gs_vmzv_tab** and **g_vmzv_ref**. Using the COMMIT construct, INSERT and UPDATE operations are committed to database. Without commit INSERT and UPDATE would be lost and not committed to database.

2.3.22. **ROLLBACK Language Construct**

The ROLLBACK language construct allows all database related operations such as INSERT, UPDATE and DELETE to be rolled back. With ROLLBACK construct, prior INSERT, UPDATE and DELETE operations are ignored and rolled back from database. Example for COMMIT construct is given below:

2.3.22.1. **ROLLBACK Construct usage**

```
data g_vmzv_tab type table of vmzv .
data g_vmzv_ref type ref to vmzv .

insert vmzv from g_vmzv_ref .
update vmzv from table g_vmzv_tab.

rollback .
```

Code Sample 2.72 *ROLLBACK Construct Sample 1*

In the example above (***Code Sample 2.72***), the **gs_vmzv_tab** internal table and **g_vmzv_ref** internal structure reference, whose types are **vmzv** table defined in the database, are defined. INSERT and UPDATE operations for table **vmzv** are performed by using **gs_vmzv_tab** and **g_vmzv_ref**. Using the ROLLBACK construct, prior INSERT and UPDATE operations are ignored and rolled back from database.

2.3.23. **DELETE Language Construct**

The DELETE language construct allows a record in a database table to be deleted by using internal structure or internal structure references. To delete multiple records from a database table, internal tables or internal table references might also be used. In latter case, **"table"** keyword must be added after **"from"** keyword.

In the DELETE construct for single record deletion, type of internal structure or internal structure reference must include all key fields of database table. Similarly, in the DELETE construct for multiple records deletion, type of internal table or internal table reference must include all key fields of database table. Examples for DELETE construct are given below:

2.3.23.1. **DELETE Construct usage with internal structure**

```
data g_vmzv type vmzv .

delete vmzv from g_vmzv.
```

Code Sample 2.73 *Delete Construct Sample 1*

In the example above (***Code Sample 2.73***), the **g_vmzv** internal structure, whose type is **vmzv** table defined in the database, is defined. Using the DELETE construct, the record in **vmzv** database table indicated by g_vmzv internal structure, would be deleted from database.

2.3.23.2. **DELETE Construct usage with internal structure reference**

```
data g_vmzv_ref type ref to vmzv .

delete vmzv from g_vmzv_ref .
```

Code Sample 2.74 *Delete Construct Sample 2*

In the example above (***Code Sample 2.74***), the **g_vmzv_ref** internal structure reference, whose type is **vmzv** table defined in the database, is defined. Using the DELETE construct, the record in **vmzv** database table indicated by data referenced by **g_vmzv_ref** internal structure reference, would be deleted from database.

2.3.23.3. **DELETE Construct usage with internal table**

```
data g_vmzv_tab type table of vmzv .

delete vmzv from table g_vmzv_tab.
```

Code Sample 2.75 *Delete Construct Sample 3*

In the example above (***Code Sample 2.75***), the **g_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, is defined. Using the DELETE construct, records in **vmzv** database table indicated by records in **g_vmzv_tab** internal table, would be deleted from database.

2.3.23.4. **DELETE Construct usage with internal table reference**

```
data g_vmzv_ref_tab type ref to table of vmzv .

delete vmzv from table g_vmzv_ref_tab.
```

Code Sample 2.76 *Delete Construct Sample 4*

In the example above (***Code Sample 2.76***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, is defined. Using the DELETE construct, records in **vmzv** database table indicated by records in table referenced by **g_vmzv_ref_tab** internal table reference, would be deleted from database.

.

2.3.24. **LOOP Language Construct**

The LOOP language construct allows you to loop through internal tables or internal table references. The `**where**` keyword can be used with the LOOP construct, so that the loop is executed only on the rows that match the filtering condition specified in the `**where**` keyword, rather than the entire internal table.

Records matching the specified criteria in the LOOP construct are processed sequentially, and the processed record is transferred to the internal structure or internal structure reference specified with the `**into**` statement. If an internal structure is specified, all data of the processed record is copied into the internal structure. If an internal structure reference is specified, the internal structure reference refers to the processed row, and no copying operation is performed. When using an internal structure reference, the internal structure reference must be defined with the `**type weak ref to**` statement. In other words, only weak references can be used with the `**into**` statement.

In LOOP construct, the object being looped through can be an **internal table** or a **reference to an internal table**. All examples below use internal tables. Even if an internal table reference is used instead of an internal table in the examples, the LOOP construct would still function as intended. Examples for LOOP construct are given below:

2.3.24.1. **LOOP Construct usage without "where" keyword and with an internal structure**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .

loop at gs_usbld_tab into gs_usbld .

endloop .
```

Code Sample 2.77 *Loop Construct Sample 1*

In the example above (***Code Sample 2.77***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, and the internal table **gs_usbld_tab**, whose type is also **usbld** table, are defined. Using LOOP construct, all records in the **gs_usbld_tab** table are

processed sequentially, and in each operation, the relevant record is copied to the **gs_usbld** internal structure.

2.3.24.2. **LOOP Construct usage without "where" keyword and with an internal structure reference**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .

loop at gs_usbld_tab into gs_usbld_ref.

endloop .
```

Code Sample 2.78 *Loop Construct Sample 2*

In the example above (***Code Sample 2.78***), a weak internal structure reference, **gs_usbld_ref**, whose type is **usbld** table defined in the database, and an internal table, **gs_usbld_tab** whose type is also **usbld** table, are defined. Using LOOP construct, all records in the **gs_usbld_tab** table are processed sequentially, and in each processing step, the **gs_usbld_ref** internal structure reference refers to the relevant record; however, no copying operation is performed.

2.3.24.3. **LOOP Construct usage with "where" keyword, internal structure in transfer and internal structure in comparison**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .

loop at gs_usbld_tab into gs_usbld
                    where sblid =  gs_vmzg-sblid .

endloop .
```

Code Sample 2.79 *Loop Construct Sample 3*

In the example above (***Code Sample 2.79***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, the internal table **gs_usbld_tab**, whose type is also **usbld** table, and the internal structure **gs_vmzg**, whose type is **vmzg** table defined in the database, are defined. Using LOOP construct, only the records from the **gs_usbld_tab** table

whose **sblid** matches the value in **gs_vmzg-sblid** are processed sequentially, and in each process, the relevant record is copied to the **gs_usbld** internal structure.

2.3.24.4. LOOP Construct usage with "where" keyword, internal structure reference in transfer and internal structure in comparison

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .

loop at gs_usbld_tab into gs_usbld_ref
                        where sblid =  gs_vmzg-sblid .

endloop .
```

Code Sample 2.80 *Loop Construct Sample 4*

In the example above (***Code Sample 2.80***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, the internal table **gs_usbld_tab**, whose type is also **usbld** table, and the internal structure **gs_vmzg**, whose type is **vmzg** table defined in the database, are defined. Using LOOP construct, only the records in the **gs_usbld_tab** table whose **sblid** matches the value in **gs_vmzg-sblid** are processed sequentially, and in each process, the **gs_usbld_ref** internal structure reference refers to the relevant record; however, no copying operation is performed.

2.3.24.5. LOOP Construct usage with "where" keyword, internal structure in transfer and parameter in comparison

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data p_string type string .

loop at gs_usbld_tab into gs_usbld
                        where sblid =  p_string .

endloop .
```

Code Sample 2.81 *Loop Construct Sample 5*

In the example above (***Code Sample 2.81***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also

usbld, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using LOOP construct, only the records from the **gs_usbld_**tab table whose **sblid** matches the value in **p_string** are processed sequentially, and in each process, the relevant record is copied to the **gs_usbld** internal structure.

2.3.24.6. **LOOP Construct usage with "where" keyword, internal structure reference in transfer and parameter in comparison**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data p_string type string .

 loop at gs_usbld_tab into gs_usbld_ref
                        where sblid =  p_string .

 endloop .
```

Code Sample 2.82 *Loop Construct Sample 6*

In the example above (***Code Sample 2.82***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is **usbld**, and the **p_string** parameter, whose type is built-in type **string**, are defined. Using LOOP construct, only the records from the **gs_usbld_tab** table whose **sblid** matches the value in **p_string** are processed sequentially, and in each process, the **gs_usbld_ref** internal structure reference refers to the relevant record; however, no copying operation is performed.

2.3.24.7. **LOOP Construct usage with "where" , "and" keywords and internal structure in transfer**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .
data p_string type string .

loop at gs_usbld_tab into gs_usbld
                        where sblid = gs_vmzg-sblid
                           and dlid =  p_string .


endloop .
```

Code Sample 2.83 *Loop Construct Sample 7*

In the example above (***Code Sample 2.83***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using LOOP construct, only the records from the **gs_usbld_tab** table whose **sblid** matches the value in **gs_vmzg-sblid** and whose **dlid** matches the value in **p_string** are processed sequentially, and in each process, the relevant record is copied to the **gs_usbld** internal structure.

2.3.24.8. **LOOP Construct usage with "where" , "and" keywords and internal structure reference in transfer**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .
data p_string type string .

 loop at gs_usbld_tab into gs_usbld_ref
                          where sblid = gs_vmzg-sblid
                             and dlid =  p_string .

 endloop .
```

Code Sample 2.84 *Loop Construct Sample 8*

In the example above (***Code Sample 2.84***), the **gs_usbld_ref** weak internal structure, reference whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using LOOP construct, only the records from the **gs_usbld_tab** table whose **sblid** matches the value in **gs_vmzg-sblid** and whose **dlid** matches the value in **p_string** are processed sequentially, and in each processing, the **gs_usbld_ref** internal structure reference refers to the relevant record; however, no copying operation is performed.

2.3.25. **READ Language Construct**

The READ language construct is quite similar to the LOOP construct. The READ construct allows only the first row matching the filtering criteria in internal tables to be processed. The "**where**" statement can be used with the READ construct, thus processing only the first row in the internal table that matches the filtering condition specified by "**where**" statement.

The first record matching the criteria specified in the READ construct is transferred to the internal structure or internal structure reference specified by the **`into`** statement. If an internal structure is specified, all data from the first record found is copied into the internal structure. If an internal structure reference is specified, the internal structure reference refers to the first record found, and no copying operation is performed. When using an internal structure reference, the internal structure reference must be defined with the "**type weak ref to**" statement. In other words, only weak references can be used with the "**into**" statement within READ construct.

In READ construct, the object being looped through can be an internal table or a reference to an internal table. All examples below use an internal table. Even if an internal table reference is used instead of an internal table in the examples, the READ construct would still work as intended.

If no record matching the criteria specified in the READ construct is found, no action is taken and the **sy-result** field is set to 0. If a record is found, the **sy-result** field is set to 1. Examples for READ construct are given below.

2.3.25.1. **READ Construct usage without "where" keyword and with internal structure**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .

read table gs_usbld_tab into gs_usbld .
```

Code Sample 2.85 *Read Construct Sample 1*

In the example above (***Code Sample 2.85***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, and the **gs_usbld_tab** internal structure, whose type is also **usbld**, are defined. The first record in the **gs_usbld_tab** table is copied to the **gs_usbld** internal structure using the READ construct.

2.3.25.2. **READ Construct usage without "where" keyword and with internal structure reference**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .

 read table gs_usbld_tab into gs_usbld_ref.
```

Code Sample 2.86 *Read Construct Sample 2*

In the example above (***Code Sample 2.86***), a weak internal structure reference, **gs_usbld_ref**, whose type is **usbld** table defined in the database and **gs_usbld_tab** internal table whose type is also **usbld** table, are defined. The **gs_usbld_ref** internal structure reference references the first record in the **gs_usbld_tab** internal table by using READ construct, but no copying operation is performed.

2.3.25.3. **READ Construct usage with "where" keyword, internal structure in transfer and internal structure in comparison**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .

read table gs_usbld_tab into gs_usbld
                        where sblid =  gs_vmzg-sblid .
```

Code Sample 2.87 *Read Construct Sample 3*

In the example above (***Code Sample 2.87***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, the internal table **gs_usbld_tab**, whose type is also **usbld** table, and the internal structure **gs_vmzg**, whose type is **vmzg** table defined in the database, are defined. Using the READ construct, only the first record from the **gs_usbld_tab** table whose **sblid** matches the value in **gs_vmzg-sblid** is copied to the **gs_usbld** internal structure.

2.3.25.4. **READ Construct usage with "where" keyword, internal structure reference in transfer and internal structure in comparison**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .

read table gs_usbld_tab into gs_usbld_ref
                        where sblid =  gs_vmzg-sblid .
```

Code Sample 2.88 *Read Construct Sample 4*

In the example above (***Code Sample 2.88***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld** table, and the **gs_vmzg** internal structure, whose type is **vmzg** table defined in the database, are defined. The **gs_usbld_ref** internal structure reference, references only the first record in the **gs_usbld_tab** table whose **sblid** is the same as the value in **gs_vmzg-sblid**; however, no copying operation is performed.

2.3.25.5. **READ Construct usage with "where" keyword, internal structure in transfer and parameter in comparison**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data p_string type string .

read table gs_usbld_tab into gs_usbld
                          where sblid =  p_string .
```

Code Sample 2.89 *Read Construct Sample 5*

In the example above (***Code Sample 2.89***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using the READ construct, only the first record from the **gs_usbld_tab** table whose **sblid** matches the value in **p_string** is copied to the **gs_usbld** internal structure.

2.3.25.6. **READ Construct usage with "where" keyword, internal structure reference in transfer and parameter in comparison**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data p_string type string .

 read table gs_usbld_tab into gs_usbld_ref
                          where sblid =  p_string .
```

Code Sample 2.90 *Read Construct Sample 6*

In the example above (***Code Sample 2.90***), the **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using the READ construct, the **gs_usbld_ref** internal structure reference, references only the first record in the **gs_usbld_tab** table whose **sblid** matches the value in the **p_string**; however, no copying operation is performed.

2.3.25.7. **READ Construct usage with “where” , “and” keyword and internal structure in transfer**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .
data p_string type string .

read table gs_usbld_tab into gs_usbld
                        where sblid = gs_vmzg-sblid
                           and dlid =  p_string .
```

Code Sample 2.91 *Read Construct Sample 7*

In the example above (***Code Sample 2.91***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using the READ construct, only the first record from the **gs_usbld_tab** table whose **sblid** is the value in **gs_vmzg-sblid** and whose **dlid** is the value in **p_string** is copied to the **gs_usbld** internal structure.

2.3.25.8. **READ Construct usage with “where” , “and” keyword and internal structure reference in transfer**

```
data gs_usbld_ref type weak ref to usbld .
data gs_usbld_tab type table of usbld .
data gs_vmzg type vmzg .
data p_string type string .

 read table gs_usbld_tab into gs_usbld_ref
                        where sblid = gs_vmzg-sblid
                           and dlid =  p_string .
```

Code Sample 2.92 *Read Construct Sample 8*

In the example above (***Code Sample 2.92***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database, the **gs_usbld_tab** internal table, whose type is also **usbld**, and the **p_string** parameter, whose type is built-in **string** type, are defined. Using the READ construct, **gs_usbld_ref** references the only first record in the **gs_usbld_tab** table

whose **sblid** is the value in **gs_vmzg-sblid** and whose **dlid** is the value **in p_string**; however, no copying operation is performed.

2.3.26. **IF-ELSE Language Construct**

The IF-ELSE language construct allows for conditional checking within a program and determines the code to be executed in the next step based on the result of this check. The IF-ELSE construct must be closed using the "**endif"** keyword after the code defined under IF-ELSE construct.

Within IF-ELSE construct, internal structures, internal tables, parameters, internal structure references and internal table references can be used. **Parentheses**, "**and**" and "**or**" keywords can be used within an IF-ELSE construct. The following 12 types of comparisons can be made using the IF-ELSE construct:

- Equality Comparison(=)
- Less than or Equal Comparison(<=)
- Greater than or Equal Comparison (>=)
- Less than Comparison(<)
- Greater than Comparison(>)
- Not Equal Comparison(**ne**)
- Includes Comparison(**in**)
- Does not Include Comparison(**not in**)
- No record exists Comparison(**is initial**)
- Records exists Comparison(**is not initial**)
- No content Comparison(**is initial**)
- Content exist Comparison(**is not initial**)

Examples for IF-ELSE construct are given below:

2.3.26.1. **IF-ELSE construct with single comparison by using internal structure and parameter**

```
data gs_usbld type usbld .
data p_string type string.

if gs_usbld-sblid = p_string .

endif.
```

Code Sample 2.93 *If-Else Construct Sample 1*

In the example above (***Code Sample 2.93***), the **gs_usbld** internal structure, whose type is **usbld** table defined in the database**, p_string** parameter whose type is built-in **string** type, are defined. Using an IF-ELSE construct, the code defined within the IF-ELSE construct is executed only if the value in **gs_usbld-sblid** is equal to the value in **p_string**.

2.3.26.2. **IF-ELSE construct with single comparison by using internal structures**

```
data gs_usbld type usbld .
data g_vmzg type vmzg.

if gs_usbld-sblid = g_vmzg-sblid.

endif.
```

Code Sample 2.94 *If-Else Construct Sample 2*

In the example above (***Code Sample 2.94***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, and the internal structure **gs_vmzg**, whose type is **vmzg** table defined in the database, are defined. The code defined within the IF-ELSE construct is executed only if the value in **gs_usbld-sblid** is equal to the value in **g_vmzg-sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison .

2.3.26.3. **IF-ELSE construct with single comparison by using internal structure and internal structure reference**

```
data gs_usbld type usbld .
data g_vmzg_ref type ref to vmzg.

if gs_usbld-sblid = g_vmzg_ref->sblid.

endif.
```

Code Sample 2.95 *If-Else Construct Sample 3*

In the example above (***Code Sample 2.95***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, and the internal structure reference **gs_vmzg_ref**, whose type is **vmzg** table defined in the database, are defined. The the code within IF-ELSE construct is executed only if the value in **gs_usbld-sblid** is equal to the value in **gs_vmzg_ref-sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison.

2.3.26.4. **IF-ELSE construct with single comparison by using internal structure and parameter**

```
data p_string type string.
data g_vmzg_ref type ref to vmzg.

if g_vmzg_ref->sblid = p_string.

endif.
```

Code Sample 2.96 *If-Else Construct Sample 4*

In the example above (***Code Sample 2.96***), the **p_string** parameter, whose type is built-in **string** type, **gs_vmzg_ref** internal structure reference, whose type is **vmzg** table defined in the database, are defined. The code defined within the IF-ELSE construct is executed only if the value in **p_string** is equal to the value in **gs_vmzg_ref->sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison.

2.3.26.5. **IF-ELSE construct with single comparison by using internal structure references**

```
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

if gs_usbld_ref->sblid = g_vmzg_ref->sblid.

endif.
```

Code Sample 2.97 *If-Else Construct Sample 5*

In the example above (***Code Sample 2.97***), the internal structure reference **gs_usbld_ref**, whose type is **usbld** table defined in the database, and the internal structure reference **gs_vmzg_ref**, whose type is **vmzg** table defined in the database, are defined. With the IF-ELSE co, ttructhe code defined inside the IF-ELSE construct is executed only if the value in **gs_usbld_ref->sblid** is equal to the value in **gs_vmzg_ref->sblid**.

2.3.26.6. **IF-ELSE construct with single comparison, internal table record exists control**

```
data gs_usbld_tab type table of usbld .

if gs_usbld_tab is initial .

endif.
```

Code Sample 2.98 *If-Else Construct Sample 6*

In the example above (***Code Sample 2.98***), gs_usbld_tab internal table, whose type is **usbld** table defined in the database, is defined. The code defined within the IF-ELSE construct is executed if there are no records in the **gs_usbld_tab** table.

In the example above, the **`is not initial`** check can also be performed instead of **`is initial`**. In this case, the code defined within the IF-ELSE construct will be executed only if there is at least one record in the **gs_usbld_tab** internal table.

2.3.26.7. **IF-ELSE construct with single comparison, internal table reference record exists control**

```
data gs_usbld_ref_tab type ref to table of usbld .

if gs_usbld_ref_tab is initial .

endif.
```

Code Sample 2.99 *If-Else Construct Sample 7*

In the example above (***Code Sample 2.99***), gs_usbld_ref_tab internal table reference, whose type is **usbld** table defined in the database, is defined. The code defined within the IF-ELSE construct is executed only if the table referenced by the **gs_usbld_ref_tab** contains no records.

In the example above, the **`is not initial`** check can also be performed instead of **`is initial`**. In this case, the code defined within the IF-ELSE construct will be executed only if the table referenced by the **gs_usbld_ref_tab** has at least one record.

2.3.26.8. **IF-ELSE construct with single comparison, internal structure content exists control**

```
data gs_usbld type usbld .

if gs_usbld is initial .

endif.
```

Code Sample 2.100 *If-Else Construct Sample 8*

In the example above (***Code Sample 2.100***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, is defined. The code defined within the IF-ELSE construct is executed only if all fields within **gs_usbld** internal structure is initial.

In the example above, the **“is not initial”** check can also be performed instead of **“is initial”**. In this case, the code defined within the IF-ELSE construct will be executed only if there is data in at least one field of the “**gs_usbld**” internal structure.

2.3.26.9. **IF-ELSE construct with single comparison, internal structure reference content exists control**

```
data gs_usbld_ref type ref to usbld .

if gs_usbld_ref is initial .

endif.
```

Code Sample 2.101 *If-Else Construct Sample 9*

In the example above (***Code Sample 2.101***), **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, is defined. The code defined within the IF-ELSE construct is only executed if all of the fields within the internal structure referenced by **gs_usbld_ref** contain no data or in other words are initial.

In the example above, the “**is not initial**” check can also be performed instead of **“is initial”**. In this case, the code defined inside the IF-ELSE construct will be executed only if at least one column of the internal structure referenced by the “**gs_usbld_ref**” contains data or in other words is not initial .

2.3.26.10. **IF-ELSE construct with multiple comparison by using “and” keyword**

```
data gs_usbld_ref_tab type ref to table of usbld .
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

if gs_usbld_ref->sblid = g_vmzg_ref->sblid and gs_usbld_ref_tab is initial.

endif.
```

Code Sample 2.102 *If-Else Construct Sample 10*

In the example above (***Code Sample 2.102***), the **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, the **gs_usbld_ref_tab** internal table reference, whose type is also **usbld** and the **gs_vmzg_ref** internal structure reference,

whose type is **vmzg** table defined in the database, are defined. The code defined within the IF-ELSE construct is executed only if the value in **gs_usbld_ref->sblid** is equal to the value in **gs_vmzg_ref->sblid** and there are no records in the table referenced by the **gs_usbld_ref_tab**.

In the example above, the **"or"** keyword can also be used instead of **"and"** keyword. In this case, the code defined inside the IF-ELSE construct will be executed only if the value in **gs_usbld_ref->sblid** is equal to the value in **gs_vmzg_ref->sblid**, or if there are no records in the table referenced by the **gs_usbld_ref_tab.**

2.3.26.11. **IF-ELSE construct usage with "else" keyword**

```
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

if gs_usbld_ref->sblid = g_vmzg_ref->sblid.

else.

endif.
```

Code Sample 2.103 *If-Else Construct Sample 11*

In the example above (***Code Sample 2.103***),the **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, the **gs_usbld_ref_tab** internal table reference, whose type is also **usbld**, and the **gs_vmzg_ref** internal structure reference, whose type is **vmzg** table defined in the database, are defined. The code defined within IF-ELSE construct is executed only if the value in **gs_usbld_ref->sblid** is equal to the value in **gs_vmzg_ref->sblid**, otherwise, the code defined under the "**else**" keyword is executed.

2.3.26.12. **IF-ELSE construct usage with “else if” keyword**

```
data gs_usbld_ref_tab type ref to table of usbld .
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

if gs_usbld_ref->sblid = g_vmzg_ref->sblid.

else if gs_usbld_ref_tab is initial .

endif.
```

Code Sample 2.104 *If-Else Construct Sample 12*

In the example above (***Code Sample 2.104***), the **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, the **gs_usbld_ref_tab** internal table reference, whose type is also **usbld**, and the **gs_vmzg_r**ef internal structure reference, whose type is **vmzg** table defined in the database, are defined. The code defined inside the IF-ELSE construct, is executed only if the value in **gs_usbld_ref->sblid** is equal to the value in **gs_vmzg_ref->sblid**. If they are not equal, the “**else if**” statement is executed, checking whether a record exists in **gs_usbld_ref_tab**. If **gs_usbld_ref_tab** does not contain any record then the result is true, so the code defined under the “**else if**” statement is executed.

2.3.27. **WHILE Language Construct**

The WHILE language construct allows for conditional checking within a program and determines the code to be executed in the next step based on the result of that check. The WHILE construct must be closed using an **"endwhile"** keyword after the code defined under the WHILE construct.

The most important difference between the WHILE construct and the IF-ELSE construct is that the WHILE construct executes the code defined within it as long as the conditional check is true. In WHILE construct, if the conditional check is true, the code defined within it is executed, and when the executed code ends, the conditional check is run again. If the result is true, the code within the WHILE construct is executed again. If the result is false, the WHILE construct is exited.

Within WHILE construct, internal structures, internal tables, parameters, internal structure references and internal table references can be used. **Parentheses**, "**and**" and "**or**" keywords can be used within a WHILE construct. The following 12 types of comparisons can be made using the WHILE construct:

- Equality Comparison(=)
- Less than or Equal Comparison(<=)
- Greater than or Equal Comparison (>=)
- Less than Comparison(<)
- Greater than Comparison(>)
- Not Equal Comparison(**ne**)
- Includes Comparison(**in**)
- Does not Include Comparison(**not in**)
- No record exists Comparison(**is initial**)
- Records exists Comparison(**is not initial**)
- No content Comparison(**is initial**)
- Content exist Comparison(**is not initial**)

Examples for WHILE construct are given below:

2.3.27.1. **WHILE construct with single comparison by using internal structure and parameter**

```
data gs_usbld type usbld .
data p_string type string.

while gs_usbld-sblid = p_string .

endwhile.
```

Code Sample 2.105 *While Construct Sample 1*

In the example above (***Code Sample 2.105***), the **gs_usbld** internal structure, whose type is the usbld table defined in the database, and **p_string** parameter whose type is built-in string type, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld-sblid** is not equal to the value in **p_string**.

2.3.27.2. **WHILE construct with single comparison by using two internal structures**

```
data gs_usbld type usbld .
data g_vmzg type vmzg.

while gs_usbld-sblid = g_vmzg-sblid.

endwhile.
```

Code Sample 2.106 *While Construct Sample 2*

In the example above (***Code Sample 2.106***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, and the internal structure **gs_vmzg**, whose type is also **vmzg** table defined in the database, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld-sblid** is not equal to the value in **g_vmzg-sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison.

2.3.27.3. **WHILE construct with single comparison by using internal structure and internal structure reference**

```
data gs_usbld type usbld .
data g_vmzg_ref type ref to vmzg.

while gs_usbld-sblid = g_vmzg_ref->sblid.

endwhile.
```

Code Sample 2.107 *While Construct Sample 3*

In the example above (***Code Sample 2.107***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, and the internal structure reference **gs_vmzg_ref**, whose type is **vmzg** table defined in the database, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld-sblid** is not equal to the value in **gs_vmzg_ref-sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison.

2.3.27.4. **WHILE construct with single comparison by using internal structure reference and parameter**

```
data p_string type string.
data g_vmzg_ref type ref to vmzg.

while g_vmzg_ref->sblid = p_string.

endwhile.
```

Code Sample 2.108 *While Construct Sample 4*

In the example above (***Code Sample 2.108***), the **p_string** parameter, whose type is built-in **string** type, the internal structure reference **gs_vmzg_ref**, whose type is also **vmzg** table defined in the database, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **p_string** is not equal to the value in **gs_vmzg_ref->sblid**.

In the example above, comparisons such as inequality (**ne**), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) can also be made instead of equality (=) comparison.

2.3.27.5. **WHILE construct with single comparison by using internal structure references**

```
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

while gs_usbld_ref->sblid = g_vmzg_ref->sblid.

endwhile.
```

Code Sample 2.109 *While Construct Sample 5*

In the example above (***Code Sample 2.109***), the internal structure reference **gs_usbld_ref**, whose type is **usbld** table defined in the database, and the internal structure reference **gs_vmzg_ref**, whose type is **vmzg** table defined in the database, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld_ref->sblid** is not equal to the value in **gs_vmzg_ref->sblid**.

2.3.27.6. **WHILE construct with single comparison, internal table record exists control**

```
data gs_usbld_tab type table of usbld .

while gs_usbld_tab is initial .

endwhile.
```

Code Sample 2.110 *While Construct Sample 6*

In the example above (***Code Sample 2.110***), **gs_usbld_tab** internal table, whose type is **usbld** table defined in the database, is created. The code defined inside the WHILE construct is executed repeatedly until there is a record in the **gs_usbld_tab** table.

In the example above, the **“is not initial”** check can also be performed instead of **“is initial”** check. In this case, the code defined inside the WHILE construct is executed repeatedly until there is no record in the **gs_usbld_tab** table.

2.3.27.7. **WHILE construct with single comparison, internal table reference record exists control**

```
data gs_usbld_ref_tab type ref to table of usbld .

while gs_usbld_ref_tab is initial .

endwhile.
```

Code Sample 2.111 *While Construct Sample 7*

In the example above (***Code Sample 2.111***), **gs_usbld_ref_tab** internal table reference, whose type is **usbld** table defined in the database, is defined. The code defined inside the WHILE construct is executed repeatedly until there is a record in the table referenced by the **gs_usbld_ref_tab**.

In the example above, the **“is not initial”** check can also be performed instead of **“is initial”** check. In this case, the code defined inside the WHILE construct is executed repeatedly until there is no record in the table referenced by the **gs_usbld_ref_tab**.

2.3.27.8. **WHILE construct with single comparison, internal structure content exists control**

```
data gs_usbld type usbld .

while gs_usbld is initial .

endwhile.
```

Code Sample 2.112 *While Construct Sample 8*

In the example above (***Code Sample 2.112***), **gs_usbld** internal structure, whose type is **usbld** table defined in the database, is defined. The code defined inside the WHILE construct is executed repeatedly until at least one of the fields of the **gs_usbld** structure contains data, in other words is not initial

In the example above, the **“is not initial”** check can also be performed instead of **“is initial”** check. In this case, the code defined inside the WHILE construct is executed repeatedly until all of the fields of the **gs_usbld** structure do not contain data, in other words are initial.

2.3.27.9. **WHILE construct with single comparison, internal structure reference content exists control**

```
data gs_usbld_ref type ref to usbld .

while gs_usbld_ref is initial .

endwhile.
```

Code Sample 2.113 *While Construct Sample 9*

In the example above (***Code Sample 2.113***), **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, is defined. The code defined inside the WHILE construct is executed repeatedly until at least one of the fields of the internal structure referenced by **gs_usbld_ref** contains data, in other words is not initial.

In the example above, the **“is not initial”** check can also be performed instead of **“is initial”** check. In this case, the code defined inside the WHILE construct is executed repeatedly until all of the fields of the internal structure referenced by **gs_usbld_ref** do not contain data, in other words are initial.

2.3.27.10. **WHILE construct with multiple comparison by using “and” keyword**

```
data gs_usbld_ref_tab type ref to table of usbld .
data gs_usbld_ref type ref to usbld .
data g_vmzg_ref type ref to vmzg.

while gs_usbld_ref->sblid = g_vmzg_ref->sblid and gs_usbld_ref_tab is initial.

endwhile.
```

Code Sample 2.114 *While Construct Sample 10*

In the example above (***Code Sample 2.114***), the **gs_usbld_ref** internal structure reference, whose type is **usbld** table defined in the database, the **gs_usbld_ref_tab** internal

table reference, whose type is also **usbld**, and the **gs_vmzg_ref** internal structure reference, whose type is **vmzg** table defined in the database, are defined. The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld_ref->sblid** is not equal to the value in **gs_vmzg_ref->sblid** or there is at least one record in the table referenced by the **gs_usbld_ref_tab**.

In the example above, the **“or”** keyword can also be used instead of **“and”** keyword. In this case, the code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld_ref->sblid** is not equal to the value in **gs_vmzg_ref->sblid**, and there is at least one record in the table referenced by the **gs_usbld_ref_tab** object .

2.3.28. **APPEND Language Construct**

The APPEND language construct allows data from an internal structure or internal structure reference to be added to an internal table or a table referenced by an internal table reference. Examples for APPEND construct are given below:

2.3.28.1. **APPEND construct usage with internal table and internal structure**

```
data gs_vmzv_tab type table of vmzv .
data lc_vmzv type vmzv.

append lc_vmzv to gs_vmzv_tab .
```

Code Sample 2.115 *Append Construct Sample 1*

In the example above (***Code Sample 2.115***), the **gs_vmzv**_tab internal table, whose type is **vmzv** table defined in the database, and the **lc_vmzv** internal structure, whose type is also **vmzv** table defined in the database, are defined. Using the APPEND construct, the **lc_vmzv** internal structure is added to the **gs_vmzv_tab** table.

2.3.28.2. **APPEND construct usage with internal table and internal structure reference**

```
data gs_vmzv_tab type table of vmzv.
data lc_vmzv_ref type ref to vmzv.

append lc_vmzv_ref to gs_vmzv_tab.
```

Code Sample 2.116 *Append Construct Sample 2*

In the example above (***Code Sample 2.116***), the internal table **gs_vmzv_tab**, whose type is **vmzv** table defined in the database, and the internal structure reference **lc_vmzv_ref**, whose type is also **vmzv** table defined in the database, are defined. Using the APPEND construct, the data referenced by the **lc_vmzv_ref** internal structure reference is added to the **gs_vmzv_tab** table.

2.3.28.3. **APPEND construct usage with internal table reference and internal structure**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv type vmzv.

append lc_vmzv to gs_vmzv_ref_tab.
```

Code Sample 2.117 *Append Construct Sample 3*

In the example above (***Code Sample 2.117***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, and the **lc_vmzv** internal structure, whose type is also **vmzv** table defined in the database, are defined. Using the APPEND construct, the **lc_vmzv** internal structure is added to the table referenced by the **gs_vmzv_ref_tab** internal table reference.

2.3.28.4. **APPEND construct usage with internal table reference and internal structure reference**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv_ref type ref to vmzv.

append lc_vmzv_ref to gs_vmzv_ref_tab.
```

Code Sample 2.118 *Append Construct Sample 4*

In the example above (***Code Sample 2.118***), the internal table reference **gs_vmzv_ref_tab**, whose type is **vmzv** table of type defined in the database, and the internal structure reference **lc_vmzv_ref**, whose type is also **vmzv**, are defined. Using the APPEND construct, the data referenced by the **lc_vmzv_ref** internal structure reference is added to the table referenced by the **gs_vmzv_ref_tab** internal table reference.

2.3.29. **MOVE-CORRESPONDING FIELDS Language Construct**

The MOVE-CORRESPONDING FIELDS language construct allows copying data from an internal structure or internal structure reference to another an internal structure or internal structure reference based on common fields.

Compiler automatically determines the common fields between types of two internal structure or internal structure references. A field is considered common between types of two internal structure or internal structure references when its name, built-in fieldtype (fixed-length character, short, int, float, string etc.) and field length is same in both types. If both types are same, then all fields are copied. Examples for MOVE-CORRESPONDING FIELDS construct are given below.

2.3.29.1. **MOVE-CORRESPONDING FIELDS construct usage with two internal structures**

```
type ubnk_ca >> bnkid type ubnk-bnkid ,
                ibnkid type ubnk-ibnkd .

type ubnkd_ca >> bnkid type ubnkd-bnkid ,
                 dlid type ubnkd-dlid ,
                 bnkism type ubnkd-bnkism .


data lc_ubnk type ubnk_ca .
data lc_ubnkd type ubnkd_ca .

move-corresponding fields of lc_ubnk to lc_ubnkd.
```

Code Sample 2.119 *Move-Correspongding Fields Construct Sample 1*

In the example above (***Code Sample 2.119***), **ubnk_ca** and **ubnkd_ca** types are defined. Within **ubnk_ca** type, **bnkid** field whose type is **ubnk-bnkid** and **ibnkid** field whose type is **ubnk-ibnkid** are defined. Within **ubnkd_ca** type, **bnkid** field whose type is **ubnkd-bnkid, dlid** field whose type is **ubnkd- dlid** and **bnkism** field whose type is **ubnkd-bnkism** are defined.

Using the MOVE-CORRESPONDING FIELDS construct, common fields between types of **lc_ubnk** and **lc_ubnkd** internal structures are copied from **lc_ubnk** internal structure to **lc_ubnkd** internal structure**.** Compiler automatically determines **bnkid** field as common fields between **ubnk_ca** and **ubnkd_ca** types since **bnkid** field has same name, built-in fieldtype (**fixed-length character**) and field length in both types.

2.3.29.2. **MOVE-CORRESPONDING FIELDS construct usage with internal structure and internal structure reference**

```
type ubnk_ca >> bnkid type ubnk-bnkid ,
                ibnkid type ubnk-ibnkd .

type ubnkd_ca >> bnkid type ubnkd-bnkid ,
                 dlid type ubnkd-dlid ,
                 bnkism type ubnkd-bnkism .


data lc_ubnk type ubnk_ca .
data lc_ubnkd_ref type ref to ubnkd_ca .

move-corresponding fields of lc_ubnk to lc_ubnkd_ref.
```

Code Sample 2.120 *Move-Correspongding Fields Construct Sample 2*

In the example above (***Code Sample 2.120***), **ubnk_ca** and **ubnkd_ca** types are defined. Within **ubnk_ca** type, **bnkid** field whose type is **ubnk-bnkid** and **ibnkid** field whose type is **ubnk-ibnkid** are defined. Within **ubnkd_ca** type, **bnkid** field whose type is **ubnkd-bnkid, dlid** field whose type is **ubnkd- dlid** and **bnkism** field whose type is **ubnkd-bnkism** are defined.

Using the MOVE-CORRESPONDING FIELDS construct, common fields between types of **lc_ubnk** internal structure and **lc_ubnkd** internal structure reference are copied from **lc_ubnk** internal structure to data referenced **by lc_ubnkd** internal structure reference**.** Compiler automatically determines **bnkid** field as common field between **ubnk_ca** and **ubnkd_ca** types since **bnkid** field has same name, built-in fieldtype (**fixed-length character**) and field length in both types.

2.3.29.3. **MOVE-CORRESPONDING FIELDS construct usage with two internal structure references**

```
type ubnk_ca >> bnkid type ubnk-bnkid ,
                ibnkid type ubnk-ibnkd .

type ubnkd_ca >> bnkid type ubnkd-bnkid ,
                 dlid type ubnkd-dlid ,
                 bnkism type ubnkd-bnkism .


data lc_ubnk_ref type ref to ubnk_ca .
data lc_ubnkd_ref type ref to ubnkd_ca .

move-corresponding fields of lc_ubnk_ref to lc_ubnkd_ref.
```

Code Sample 2.121 *Move-Correspongding Fields Construct Sample 3*

In the example above (***Code Sample 2.121***), **ubnk_ca** and **ubnkd_ca** types are defined. Within **ubnk_ca** type, **bnkid** field whose type is **ubnk-bnkid** and **ibnkid** field whose type is **ubnk-ibnkid** are defined. Within **ubnkd_ca** type, **bnkid** field whose type is **ubnkd-bnkid, dlid** field whose type is **ubnkd- dlid** and **bnkism** field whose type is **ubnkd-bnkism** are defined.

Using the MOVE-CORRESPONDING FIELDS construct, common fields between types of **lc_ubnk** and **lc_ubnkd** internal structure references are copied from data referenced by **lc_ubnk** internal structure reference to data referenced **by lc_ubnkd** internal structure reference**.** Compiler automatically determines **bnkid** field as common field between **ubnk_ca** and **ubnkd_ca** types since **bnkid** field has same name, built-in fieldtype (**fixed-length character**) and field length in both types.

2.3.30. **CONVERT Language Construct**

The CONVERT language construct allows converting data from one built-in fieldtype to another built-in fieldtype such as from **string** to **date** or **string** to numeric built-in fieldtypes such as **double** or **int**. Parameters, internal structure fields and internal structure reference fields can be used with CONVERT construct. Examples for CONVERT construct are given below:

2.3.30.1. **CONVERT construct usage from string to numeric fields**

```
data lc_string type string .
data lc_short type short .
data lc_int type int.

data lc_float type float .
data lc_double type double.

lc_string = “14” .

convert string lc_string into number lc_short .
convert string lc_string into number lc_int.

lc_string = “255.76” .

convert string lc_string into number lc_float.
convert string lc_string into number lc_double.
```

Code Sample 2.122 *Convert Construct Sample 1*

In the example above (***Code Sample 2.122***), **lc_string** parameter whose type is built-in **string** type, **lc_short** parameter whose type is built-in **short** type, **lc_int** parameter whose type is built-in **int** type, **lc_float** parameter whose type is built-in **float** type and **lc_double** parameter whose type is built-in **double** type are defined.

The **lc_string** parameter is assigned to “**14**” number string. Using CONVERT construct, **lc_string** parameter is converted to number then assigned to **lc_short** and **lc_int** parameters. The **lc_string** parameter is then assigned to “**255.76**” decimal string. Using CONVERT construct, **lc_string** parameter is converted to decimal number then assigned to **lc_float** and **lc_double** parameters.

2.3.30.2. **CONVERT construct usage from string to date time fields**

```
data lc_string type string .

data lc_date type date .
data lc_time type time.
data lc_timestamp type timestamp.

lc_string = “12.02.2001” .
convert string lc_string into date lc_date.

lc_string = “14:30:21” .
convert string lc_string into time lc_time.

lc_string = “12.02.2001 14:30:21” .
convert string lc_string into timestamp lc_timestamp.
```

Code Sample 2.123 *Convert Construct Sample 2*

In the example above (***Code Sample 2.123***), **lc_string** parameter whose type is built-in **string** type, **lc_date** parameter whose type is built-in **date** type, **lc_time** parameter whose type is built-in **time** type and **lc_timestamp** parameter whose type is built-in **timestamp** type are defined.

The **lc_string** parameter is assigned to “**12.02.2001**” date string. Using CONVERT construct, **lc_string** parameter is converted to date then assigned to **lc_date** parameter. The **lc_string** parameter is then assigned to “**14:30:21**” time string. Using CONVERT construct, **lc_string** parameter is converted to time then assigned to **lc_time** parameter.

Finally, the **lc_string** parameter is assigned to “**12.02.2001 14:30:21**” timestamp string. Using CONVERT construct, **lc_string** parameter is converted to timestamp then assigned to **lc_timestamp** parameter.

2.3.30.3. **CONVERT construct usage from numeric fields to string**

```
data lc_string type string .
data lc_short type short .
data lc_int type int.

data lc_float type float .
data lc_double type double.

lc_short =  14.
convert number lc_short into string lc_string .

lc_int =  22 .
convert number lc_int into string lc_string .

lc_float = 143.22 .
convert number lc_float into string lc_string .

lc_double = 298.76 .
convert number lc_double into string lc_string .
```

Code Sample 2.124 *Convert Construct Sample 3*

In the example above (***Code Sample 2.124***), **lc_string** parameter whose type is built-in **string** type, **lc_short** parameter whose type is built-in **short** type, **lc_int** parameter whose type is built-in **int** type, **lc_float** parameter whose type is built-in **float** type and **lc_double** parameter whose type is built-in **double** type are defined.

The **lc_short** parameter is assigned to number **14**. Using CONVERT construct, **lc_short** parameter is converted to string then assigned to **lc_string** parameter. The **lc_int** parameter is assigned to number **22**. Using CONVERT construct, **lc_int** parameter is converted to string then assigned to **lc_string** parameter.

The **lc_float** parameter is assigned to number **143.22**. Using CONVERT construct, **lc_float** parameter is converted to string then assigned to **lc_string** parameter. Finally, the **lc_double** parameter is assigned to number **298.76**. Using CONVERT construct, **lc_double** parameter is converted to string then assigned to **lc_string** parameter.

2.3.30.4. **CONVERT construct usage from date time fields to string**

```
data lc_string type string .

data lc_date type date .
data lc_time type time.
data lc_timestamp type timestamp.

lc_date = “12.02.2001” .
convert date lc_date into string lc_string.

lc_time = “14:30:21” .
convert time lc_time into string  lc_string.

lc_timestamp = “12.02.2001 14:30:21” .
convert timestamp lc_timestamp into string lc_string.
```

Code Sample 2.125 *Convert Construct Sample 4*

In the example above (***Code Sample 2.125***), **lc_string** parameter whose type is built-in **string** type, **lc_date** parameter whose type is built-in **date** type, **lc_time** parameter whose type is built-in **time** type and **lc_timestamp** parameter whose type is built-in **timestamp** type are defined.

The **lc_date** parameter is assigned to date “**12.02.2001**”. Using CONVERT construct, **lc_date** parameter is converted to string then assigned to **lc_string** parameter. The **lc_time** parameter is assigned to time “**14:30:21**”. Using CONVERT construct, **lc_time** parameter is converted to string then assigned to **lc_string** parameter.

Finally, the **lc_timestamp** parameter is assigned to timestamp “**12.02.2001 14:30:21**”. Using CONVERT construct, **lc_timestamp** parameter is converted to string then assigned to **lc_string** parameter.

2.3.31. **GET ENTRY Language Construct**

The GET ENTRY language construct allows a record to be transferred from an internal table or a table referenced by an internal table reference into an internal structure, by specifying its index. The GET ENTRY construct also allows an internal structure reference to refer a record of an internal table or of a table referenced by an internal table reference by specifying its index.

In the GET ENTRY construct, type of internal table or internal table reference must be same with internal stucture or internal structure reference. When using an internal structure reference in the GET ENTRY construct, the internal structure reference must be defined with the **"type weak ref to"** statement. In other words, only weak references can be used with the GET ENTRY construct. Examples for GET ENTRY construct are given below:

2.3.31.1. **GET ENTRY construct usage with internal table, internal structure and parameter**

```
data gs_vmzv_tab type table of vmzv .
data lc_vmzv type vmzv.
data row_id type unsigned int.

get entry from gs_vmzv_tab into lc_vmzv index row_id.
```

Code Sample 2.126 *Get Entry Construct Sample 1*

In the example above (***Code Sample 2.126***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **row_id** parameter in the **gs_vmzv_tab** internal table is transferred to the **lc_vmzv** internal structure.

2.3.31.2. **GET ENTRY construct usage with internal table, internal structure and internal structure field**

```
data gs_vmzv_tab type table of vmzv .
data lc_vmzv type vmzv.
data lc_index type index.

get entry from gs_vmzv_tab into lc_vmzv index lc_index-row_id
```

Code Sample 2.127 *Get Entry Construct Sample 2*

In the example above (***Code Sample 2.127***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **lc_index-rowid** field in the **gs_vmzv_tab** internal table is transferred to the **lc_vmzv** internal structure.

2.3.31.3. **GET ENTRY construct usage with internal table, internal structure reference and parameter**

```
data gs_vmzv_tab type table of vmzv.
data lc_vmzv_ref type weak ref to vmzv.
data row_id type unsigned int.

get entry from gs_vmzv_tab into lc_vmzv_ref index row_id.
```

Code Sample 2.128 *Get Entry Construct Sample 3*

In the example above (***Code Sample 2.128***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv-ref** internal structure reference, whose type is also **vmzv**, and the **row_id** parameter, whose type built-in **unsigned int** type, are defined. The GET ENTRY construct ensures that the record at the index indicated by the **row_id** parameter in the **gs_vmzv_tab** internal table is referenced by the **lc_vmzv_ref** internal structure reference.

2.3.31.4. **GET ENTRY construct usage with internal table, internal structure reference and internal structure field**

```
data gs_vmzv_tab type table of vmzv.
data lc_vmzv_ref type weak ref to vmzv.
data lc_index type index.

get entry from gs_vmzv_tab into lc_vmzv_ref index lc_index-row_id .
```

Code Sample 2.129 *Get Entry Construct Sample 4*

In the example above (***Code Sample 2.129***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. The GET ENTRY construct ensures that the record at the index indicated by the **lc_index-row_id** parameter in the **gs_vmzv_tab** internal table is referenced by the **lc_vmzv_ref** internal structure reference.

2.3.31.5. **GET ENTRY construct usage with internal table reference, internal structure and parameter**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv type vmzv.
data row_id type unsigned int.

get entry from gs_vmzv_ref_tab into lc_vmzv index row_id.
```

Code Sample 2.130 *Get Entry Construct Sample 5*

In the example above (***Code Sample 2.130***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **row_id** parameter in the **gs_vmzv_ref_tab** internal table reference is transferred to the **lc_vmzv** internal structure.

2.3.31.6. **GET ENTRY construct usage with internal table reference, internal structure and internal structure field**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv type vmzv.
data lc_index type index.

get entry from gs_vmzv_ref_tab into lc_vmzv index lc_index-row_id.
```

Code Sample 2.131 *Get Entry Construct Sample 6*

In the example above (***Code Sample 2.131***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **lc_index-row_id** field in the table referenced by **gs_vmzv_tab_ref** is transferred to the **lc_vmzv** internal structure.

2.3.31.7. **GET ENTRY construct usage with internal table reference, internal structure reference and parameter**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv_ref type weak ref to vmzv.
data row_id type unsigned int.

get entry from gs_vmzv_ref_tab into lc_vmzv_ref index row_id.
```

Code Sample 2.132 *Get Entry Construct Sample 7*

In the example above (***Code Sample 2.132***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **row_id** parameter in the table referenced by **gs_vmzv_ref_tab**, is referenced by the **lc_vmzv_ref** internal structure reference.

2.3.31.8. **GET ENTRY construct usage with internal table reference, internal structure reference and internal structure field**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv_ref type weak ref to vmzv.
data lc_index type index.

get entry from gs_vmzv_ref_tab into lc_vmzv_ref index lc_index-row_id.
```

Code Sample 2.133 *Get Entry Construct Sample 8*

In the example above (***Code Sample 2.133***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using GET ENTRY construct, the record at the index indicated by the **lc_index-row_id** parameter in the table referenced by **gs_vmzv_tab_ref**, is referenced by the **lc_vmzv_ref** internal structure reference.

2.3.32. **MODIFY INDEX Language Construct**

The MODIFY INDEX language construct allows a record in an internal table or a table referenced by an internal table reference to be modified with data from an internal structure, by specifying its index. The MODIFY INDEX construct also a record in an internal table or a table referenced by an internal table reference to be modified with data referenced by an internal structure reference, by specifying its index.

In the MODIFY INDEX construct, type of internal table or internal table reference must be same with internal stucture or internal structure reference. Examples for MODIFY INDEX construct are given below:

2.3.32.1. **MODIFY INDEX construct usage with internal table, internal structure and parameter**

```
data gs_vmzv_tab type table of vmzv .
data lc_vmzv type vmzv.
data row_id type unsigned int.

modify gs_vmzv_tab from lc_vmzv index row_id.
```

Code Sample 2.134 *Modify Index Construct Sample 1*

In the example above (***Code Sample 2.134***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **row_id** parameter in the **gs_vmzv_tab** internal table is modified with the data of the **lc_vmzv** internal structure.

2.3.32.2. **MODIFY INDEX construct usage with internal table, internal structure and internal structure field**

```
data gs_vmzv_tab type table of vmzv .
data lc_vmzv type vmzv.
data lc_index type index .

modify gs_vmzv_tab from lc_vmzv index lc_index-row_id.
```

Code Sample 2.135 *Modify Index Construct Sample 2*

In the example above (***Code Sample 2.135***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **lc_index-row_id** field in the **gs_vmzv_tab** internal table is modified with the data of the **lc_vmzv** internal structure.

2.3.32.3. **MODIFY INDEX construct usage with internal table, internal structure reference and parameter**

```
data gs_vmzv_tab type table of vmzv.  refr
data lc_vmzv_ref type ref to vmzv.
data row_id type unsigned int.

modify gs_vmzv_tab from lc_vmzv_ref index row_id.
```

Code Sample 2.136 *Modify Index Construct Sample 3*

In the example above (***Code Sample 2.136***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **row_id** parameter in the **gs_vmzv_tab** internal table is modified with the data referenced by the **lc_vmzv_ref** internal structure reference.

2.3.32.4. **MODIFY INDEX construct usage with internal table, internal structure reference and internal structure field**

```
data gs_vmzv_tab type table of vmzv.
data lc_vmzv_ref type ref to vmzv.
data lc_index type index .


modify gs_vmzv_tab from lc_vmzv_ref index lc_index-row_id.
```

Code Sample 2.137 *Modify Index Construct Sample 4*

In the example above (***Code Sample 2.137***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **lc_index-row_id** field in the **gs_vmzv_tab** internal table is modified with the data referenced by the **lc_vmzv_ref** internal structure reference.

2.3.32.5. **MODIFY INDEX construct usage with internal table reference, internal structure and parameter**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv type vmzv.
data row_id type unsigned int.

modify gs_vmzv_ref_tab from lc_vmzv index row_id.
```

Code Sample 2.138 *Modify Index Construct Sample 5*

In the example above ***(Code Sample 2.138***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **row_id** parameter in the table referenced by **gs_vmzv_ref_tab** internal table is modified with the data of the **lc_vmzv** internal structure.

2.3.32.6. **MODIFY INDEX construct usage with internal table reference, internal structure and internal structure field**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv type vmzv.
data lc_index type index .

modify gs_vmzv_ref_tab from lc_vmzv index lc_index–row_id.
```

Code Sample 2.139 *Modify Index Construct Sample 6*

In the example above (***Code Sample 2.139***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv** internal structure, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **lc_index-row_id** field in the table referenced by **gs_vmzv_ref_tab** internal table is modified with the data of **lc_vmzv** internal structure.

2.3.32.7. **MODIFY INDEX construct usage with internal table reference, internal structure reference and parameter**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv_ref type ref to vmzv.
data row_id type unsigned int.

modify gs_vmzv_ref_tab from lc_vmzv_ref index row_id.
```

Code Sample 2.140 *Modify Index Construct Sample 7*

In the example above (***Code Sample 2.140***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **row_id** parameter in the table referenced by **gs_vmzv_ref_tab** internal table is modified with the data referenced by **lc_vmzv_ref** internal structure reference.

2.3.32.8. **MODIFY INDEX construct usage with internal table reference, internal structure reference and internal structure field**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_vmzv_ref type ref to vmzv.
data lc_index type index.

modify gs_vmzv_ref_tab from lc_vmzv_ref index lc_index-row_id.
```

Code Sample 2.141 *Modify Index construct Sample 8*

In the example above (***Code Sample 2.141***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database, the **lc_vmzv_ref** internal structure reference, whose type is also **vmzv**, and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the MODIFY INDEX construct, the record at the index indicated by the **lc_index-row_id** field in the table referenced by **gs_vmzv_ref_tab** internal table is modified with the data referenced by **lc_vmzv_ref** internal structure reference.

2.3.33. **DELETE INDEX Language Construct**

The DELETE INDEX language construct allows a record in an internal table or a table referenced by an internal table reference to be deleted by specifying its index. Examples for DELETE INDEX construct are given below:

2.3.33.1. **DELETE INDEX construct usage with internal table and parameter**

```
data gs_vmzv_tab type table of vmzv .
data row_id type unsigned int.

delete gs_vmzv_tab index row_id.
```

Code Sample 2.142 *Delete Index Construct Sample 1*

In the example above (***Code Sample 2.142***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the DELETE INDEX construct, the record at the index indicated by the **row_id** parameter in the **gs_vmzv_tab** internal table is deleted from **gs_vmzv_tab** internal table**.**

2.3.33.2. **DELETE INDEX construct usage with internal table and internal structure field**

```
data gs_vmzv_tab type table of vmzv . .
data lc_index type index .

delete gs_vmzv_tab index lc_index-row_id.
```

Code Sample 2.143 *Delete Index Construct Sample 2*

In the example above (***Code Sample 2.143***), the **gs_vmzv_tab** internal table, whose type is **vmzv** table defined in the database and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the DELETE INDEX construct, the record at the index indicated by the **lc_index-row_id** field in the **gs_vmzv_tab** internal table is deleted from **gs_vmzv_tab** internal table.

2.3.33.3. **DELETE INDEX construct usage with internal table reference and parameter**

```
data gs_vmzv_ref_tab type ref to table of vmzv..
data row_id type unsigned int.

delete gs_vmzv_ref_tab index row_id.
```

Code Sample 2.144 *Delete Index Construct Sample 3*

In the example above (***Code Sample 2.144***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database and the **row_id** parameter, whose type is built-in **unsigned int** type, are defined. Using the DELETE INDEX construct, the record at the index indicated by the **row_id** parameter in the table referenced by **gs_vmzv_ref_tab** internal table reference is deleted from the table referenced by **gs_vmzv_ref_tab** internal table reference.

2.3.33.4. **DELETE INDEX construct usage with internal table reference and internal structure field**

```
data gs_vmzv_ref_tab type ref to table of vmzv.
data lc_index type index .

delete gs_vmzv_ref_tab index lc_index–row_id.
```

Code Sample 2.145 *Delete Index Construct Sample 4*

In the example above (***Code Sample 2.145***), the **gs_vmzv_ref_tab** internal table reference, whose type is **vmzv** table defined in the database and the **lc_index** internal structure, whose type is program-defined **index** type, are defined. Using the DELETE INDEX construct, the record at the index indicated by the **lc_index-row_id** field the table referenced by **gs_vmzv_ref_tab** internal table reference is deleted from the table referenced by **gs_vmzv_ref_tab** internal table reference.

2.3.34. **CONCATENATE Language Construct**

The CONCATENATE language construct allows to concenate internal structure fields, internal structure reference fields, parameters and constants into internal structure field or parameter. If numeric values used in CONCATENATE construct, then these numeric values are converted into character strings before they are concatenated.

If internal structure field is used as concatenation target, then this field should be character type field either fixed length character field or built-in **string** type. In case of concatenation target being fixed length character field and concatenated sources are bigger than fixed lenght, then result is truncated to fixed length. If parameter is used as concatenation target, then type of this parameter should be built-in **string** type.

Using CONCATENATE construct, any number of strings or numeric values can be concatenated. Examples for CONCATENATE construct are given below:

2.3.34.1. **CONCATENATE construct with internal structure fields as source and internal structure field as target**

```
data g_sthk type sthk.
data g_vmzv type vmzv .
data lc_sklt type sklt .

concatenate g_sthk-shbid, g_vmzv-mzid into lc_sklt-skltid.
```

Code Sample 2.146 *Concatenate Construct Sample 1*

In the example above (***Code Sample 2.146***), the **g_sthk** internal structure, whose type is **sthk** table defined in the database, the **g_vmzv** internal structure, whose type is **vmzv** table defined in the database and the **lc_sklt** internal structure, whose type is **sklt** table defined in the database, are defined. Using the CONCATENATE construct, internal structure fields **g_sthk-shbid**, **g_vmzv-mzid** are concatenated into **lc_sklt-skltid** internal structure field.

In the example, internal structure reference fields can also be used instead of internal structure fields. In this case, "**->**" operator should be used instead of "**-**" operator to accces internal structure reference fields.

2.3.34.2. **CONCATENATE construct with internal structure field and parameter as source and internal structure field as target**

```
data g_sthk type sthk.
data lc_mzid type string .
data lc_sklt type sklt .

concatenate g_sthk-shbid, lc_mzid into lc_sklt-skltid.
```

Code Sample 2.147 *Concatenate Construct Sample 2*

In the example above (***Code Sample 2.147***), the **g_sthk** internal structure, whose type is **sthk** table defined in the database, the **lc_mzid** parameter, whose type is built-in **string** type**,** and the **lc_sklt** internal structure, whose type is **sklt** table defined in the database, are defined. Using the CONCATENATE construct, internal structure field **g_sthk-shbid** and parameter **lc_mzid** are concatenated into **lc_sklt-skltid** internal structure field.

In the example, internal structure reference fields can also be used instead of internal structure fields. In this case, "->" operator should be used instead of "-" operator to accces internal structure reference fields.

2.3.34.3. **CONCATENATE construct with parameters as source and internal structure field as target**

```
data lc_shbid type string.
data lc_mzid type string .
data lc_sklt type sklt .

concatenate lc_shbid, lc_mzid into lc_sklt-skltid.
```

Code Sample 2.148 *Concatenate Construct Sample 3*

In the example above (***Code Sample 2.148***), the **lc_shbid** and **lc_mzid** parameters, whose type are built-in **string** type**,** and the **lc_sklt** internal structure, whose type is **sklt** table defined in the database, are defined. Using the CONCATENATE construct, parameters **lc_shbid** and **lc_mzid** are concatenated into **lc_sklt-skltid** internal structure field.

In the example, internal structure reference fields can also be used instead of internal structure fields. In this case, "->" operator should be used instead of "-" operator to accces internal structure reference fields.

2.3.34.4. **CONCATENATE construct with internal structure fields as source and parameter as target**

```
data g_sthk type sthk.
data g_vmzv type vmzv .
data lc_lock_entry type string .

concatenate g_sthk-shbid, g_vmzv-mzid into lc_lock_entry.
```

Code Sample 2.149 *Concatenate Construct Sample 4*

In the example above (***Code Sample 2.149***), the **g_sthk** internal structure, whose type is **sthk** table defined in the database, the **g_vmzv** internal structure, whose type is **vmzv** table defined in the database and the **lc_lock_entry** parameter, whose type is built-in **string** type, are defined. Using the CONCATENATE construct, internal structure fields **g_sthk-shbid**, **g_vmzv-mzid** are concatenated into **lc_lock_entry** parameter.

In the example, internal structure reference fields can also be used instead of internal structure fields. In this case, "->" operator should be used instead of "-" operator to accces internal structure reference fields.

2.3.34.5. **CONCATENATE construct with internal structure field and parameter as source and parameter as target**

```
data g_sthk type sthk.
data lc_mzid type string .
data lc_lock_entry type string .

concatenate g_sthk-shbid, lc_mzid into lc_lock_entry.
```

Code Sample 2.150 *Concatenate Construct Sample 5*

In the example above (***Code Sample 2.150***), the **g_sthk** internal structure, whose type is **sthk** table defined in the database, **lc_mzid** and **lc_lock_entry** parameters, whose type are

built-in **string** type, are defined. Using the CONCATENATE construct, internal structure field **g_sthk-shbid** and parameter **lc_mzid** are concatenated into **lc_lock_entry** parameter.

In the example, internal structure reference field can also be used instead of internal structure field. In this case, "->" operator should be used instead of "-" operator to accces internal structure reference fields.

2.3.34.6. **CONCATENATE construct with parameters as source and parameter as target**

```
data lc_shbid type string.
data lc_mzid type string.
data lc_lock_entry type string .

concatenate lc_shbid, lc_mzid into lc_lock_entry.
```

Code Sample 2.151 *Concatenate Construct Sample 6*

In the example above (***Code Sample 2.151***), **lc_shbid**, **lc_mzid** and **lc_lock_entry** parameters, whose type are built-in **string** type, are defined. Using the CONCATENATE construct, parameters **lc_shbid** and **lc_mzid** are concatenated into **lc_lock_entry** parameter.

2.3.34.7. **CONCATENATE construct with using constant as source**

```
constant lock_tab_stock_mov value "STHB"

data g_sthk type sthk.
data lc_lock_entry type string .

concatenate lock_tab_stock_mov, g_sthk-shbid into lc_lock_entry.
```

Code Sample 2.152 *Concatenate Construct Sample 7*

In the example above (***Code Sample 2.152***), constant **lock_tab_stock_mov** whose value is "STHB"**,** the **g_sthk** internal structure, whose type is **sthk** table defined in the database and the **lc_lock_entry** parameter, whose type is built-in **string** type**,** are defined. Using the CONCATENATE construct, constant **lock_tab_stock_mov** and internal structure field **g_sthk-shbid** are concatenated into **lc_lock_entry** parameter..

2.3.34.8. **CONCATENATE construct with using more than two sources**

```
constant stock_mov_doc value "Stock Movement Document "
constant not_created value " was not created"

data g_sthk type sthk.
data lc_message type string .

concatenate stock_mov_doc, g_sthk-shbid, not_created into lc_message .
```

Code Sample 2.153 *Concatenate Construct Sample 8*

In the example above (***Code Sample 2.153***), constants **stock_mov_doc** and **not_created,** the **g_sthk** internal structure, whose type is **sthk** table defined in the database and the **lc_message** parameter, whose type is built-in **string** type**,** are defined. Using the CONCATENATE construct, constant **lock_tab_stock_mov**, internal structure field **g_sthk-shbid** and constant **not_created,** are concatenated into **lc_message** parameter.

2.3.35. **CLEAR Language Construct**

The CLEAR language construct allows the clearing the contents of internal structures or contents of data referenced by internal structure references. CLEAR construct has one input parameter that is either internal structure or internal structure reference .Examples for CLEAR construct are given below:

2.3.35.1. **CLEAR construct usage with internal structure**

```
data lc_sthk type sthk.

clear lc_sthk.
```

Code Sample 2.154 *Clear Construct Sample 1*

In the example above (***Code Sample 2.154***), the **lc_sthk** internal structure whose type is **sthk** table defined in the database, is defined. Using the CLEAR construct, contents of **lc_sthk** internal structure is cleared.

2.3.35.2. **CLEAR construct usage with internal structure reference**

```
data lc_sthk_ref type ref to sthk.

clear lc_sthk_ref .
```

Code Sample 2.155 *Clear Construct Sample 1*

In the example above (***Code Sample 2.155***), the **lc_sthk_ref** internal structure reference whose type is **sthk** table defined in the database, is defined. Using the CLEAR construct, contents of data referenced by **lc_sthk_ref** internal structure reference is cleared.

2.3.36. **REFRESH Language Construct**

The REFRESH language construct allows the deleting all rows of internal tables or all rows of table referenced by internal table references. REFRESH construct has one input parameter that is either internal table or internal table reference .Examples for REFRESH construct are given below:

2.3.36.1. **REFRESH construct usage with internal table**

```
data lc_sthk_tab type table of sthk.

refresh lc_sthk_tab.
```

Code Sample 2.156 *Refresh Construct Sample 1*

In the example above (***Code Sample 2.156***), the **lc_sthk_tab** internal table whose type is **sthk** table defined in the database, is defined. Using the REFRESH construct, all rows of **lc_sthk_tab** internal table is deleted.

2.3.36.2. **REFRESH construct usage with internal table reference**

```
data lc_sthk_tab_ref type ref to table of sthk.

refresh lc_sthk_tab_ref.
```

Code Sample 2.157 *Refresh Construct Sample 1*

In the example above (***Code Sample 2.157***), the **lc_sthk_tab_ref** internal table referemce whose type is **sthk** table defined in the database, is defined. Using the REFRESH construct, all rows of table referenced by **lc_sthk_tab_ref** internal table is deleted.

2.3.37. **MESSAGE Language Construct**

The MESSAGE language construct allows program messages to be displayed to the user. MESSAGE construct has two input parameters. First input parameter of MESSAGE construct is text of message; second input parameter is message type. Both of the parameters should have to be in character type either built-in **string** type or fixed length character field. Internal structure fields, internal structure reference fields or parameters with **string** type can be used as input parameters for MESSAGE construct.

Message types that is known by runtime inclıde information message type ("**I**"), warning message type ("**W**"), and error message type ("**E**"). Examples for MESSAGE construct are given below:

2.3.37.1. **MESSAGE construct usage with two internal structure fields**

```
type message >> message_text type string,
                message_type type char .

data lc_message type message .

message  lc_message-message_text type lc_message-message_type .
```

Code Sample 2.158 *Message Construct Sample 1*

In the example above (***Code Sample 2.158***), the **message** type which includes **message_text** and **message_type** fields and **lc_message** internal structure whose type is **message,** are defined. Using the MESSAGE construct, message whose type is **lc_message-message_type,** is displayed to the end user by using text **lc_message-message_text**.

In the example above, **message_type** field of message type can be in built-in **string** type instead of built-in **char** type.

2.3.37.2. **MESSAGE construct usage with two parameters**

```
data lc_message type string .
data lc_message_type type char .

message lc_message type lc_message_type.
```

Code Sample 2.159 *Message Construct Sample 3*

In the example above (***Code Sample 2.159***), **lc_message** parameter whose type is built-in **string** type and **lc_message_type** parameter whose type is built-in **char** type**,** are defined. Using the MESSAGE construct, message whose type is **lc_message_type** is displayed to the end user by using text **lc_message.**

In the example above, **lc_message_type** can be in built-in **string** type instead of built-in **char** type.

2.3.38. **BREAK Language Construct**

The BREAK language construct allows breaking loops defined by WHILE or LOOP language constructs. Examples for BREAK construct are given below:

2.3.38.1. **BREAK Construct usage in WHILE Construct**

```
data gs_usbld type usbld .
data p_string type string.
data lc_count type unsigned short.

lc_count = 0
while gs_usbld-sblid = p_string .

  if lc_count = 10.
    break .
  endif.

 lc_count = lc_count + 1 .

endwhile.
```

Code Sample 2.160 *Break Construct Sample 1*

In the example above (***Code Sample 2.160***), the **gs_usbld** internal structure, whose type is the usbld table defined in the database, **p_string** parameter whose type is built-in **string** type, **lc_count** parameter whose type is built-in **unsigned short** type, are defined are defined.

The code defined inside the WHILE construct is executed repeatedly until the value in **gs_usbld-sblid** is not equal to the value in **p_string** and **lc_count** is equal to 10. The WHILE cosntruct is broken by using BREAK construct when value of **lc_count** reaches 10.

2.3.38.2. **BREAK Construct usage in LOOP Construct**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data lc_count type unsigned short.

lc_count = 0

 loop at gs_usbld_tab into gs_usbld .

  if lc_count = 10.
    break .
  endif.

 lc_count = lc_count + 1 .

 endloop .
```

Code Sample 2.161 *Break Construct Sample 2*

In the example above (***Code Sample 2.161***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, the internal table **gs_usbld_tab**, whose type is also **usbld** table and **lc_count** parameter whose type is built-in **unsigned short** type are defined.

Using LOOP construct, all records in the **gs_usbld_tab** table are processed sequentially until **lc_count** is equal to 10, and in each operation, the relevant record is copied to the **gs_usbld** internal structure. The LOOP construct is broken by using BREAK construct when value of **lc_count** reaches 10.

2.3.39. **CONTINUE Language Construct**

The CONTINUE language construct halts processing of current record and continues with the processing of next record in loops defined by LOOP language constructs. Example for CONTINUE construct is given below:

2.3.39.1. **CONTINUE Construct usage in LOOP Construct**

```
data gs_usbld type usbld .
data gs_usbld_tab type table of usbld .
data lc_count type unsigned short.

lc_count = 0

 loop at gs_usbld_tab into gs_usbld .

  if lc_count = 10.
    lc_count = lc_count + 1 .
    continue .
  endif.

 lc_count = lc_count + 1 .

 endloop .
```

Code Sample 2.162 *Continue Construct Sample 1*

In the example above (***Code Sample 2.162***), the internal structure **gs_usbld**, whose type is **usbld** table defined in the database, the internal table **gs_usbld_tab**, whose type is also **usbld** table and **lc_count** parameter whose type is built-in **unsigned short** type are defined.

Using LOOP construct, all records in the **gs_usbld_tab** table are processed sequentially and in each operation, the relevant record is copied to the **gs_usbld** internal structure but if **lc_count** is equal to 10, **lc_count** is incremented by 1, processing of current record is halted and the code continues with processing of next record.

2.3.40. **RETURN Language Construct**

The routines defined with ROUTINE construct, by default, terminate their processing and return after last instruction defined within routine is processed. The RETURN language construct terminates the processing the routine and returns although last instruction within routine is not reached. Example for RETURN construct is given below:

2.3.40.1. **RETURN Construct usage**

```
routine on_change_istid .

  data lc_uist like uist .
  data lc_sthk type weak ref to sthk_ca.

  data lc_message type string .
  data lc_message_type  type string .

  select single * from  uist  into corresponding
                              fields of  lc_uist
                              where istid = gd_sthbk-istid .
  if sy-result = 0 .

    lc_message = "Movement type is not defined" .
    lc_message_type = "E" .
    message lc_message type lc_message_type .

    return .

  endif.

  append lc_uist to g_uist_tab.
  refresh frame sthr_ch_details .

endroutine .
```

Code Sample 2.163 *Continue Construct Sample 1*

In the example above (***Code Sample 2.163***), routine **on_change_istid** that has no parameters, is defined. The routine is terminated before reaching last instruction if there is no record in database table **uist** whose **istid** field value is equal to value in **gd_sthbk-istid** by using RETURN construct after giving message to user by using MESSAGE construct .

2.3.41. **DISPLAY FRAME Language Construct**

The DISPLAY FRAME language construct display frames defined with FRAME user interface component. DISPLAY FRAME construct takes only one parameter that is the frame name. DISPLAY FRAME construct can only be used within routines.

DISPLAY FRAME construct calls ON-INIT function of FRAME user interface component before displaying frame if it is defined. Example for DISPLAY FRAME construct is given below:

2.3.41.1. **DISPLAY FRAME construct usage**

```
beginof frame ubnk_grid .

   top  :: 0   left :: 0     width :: 1366  height :: 744  scale:: true on-init :: init_ubnk_grid.

   toolbar init_toolbar .

      top :: 0  left :: 0  width :: 1366 height :: 35 .

      tool back >>label :: "Geri"   image ::  back on-click :: on_exit_program  .
      tool save >>label :: "Kaydet"   image ::  save on-click :: on_save  .

   endof toolbar .

   tablectrl  bank_data .

     left :: 0    top :: 25    width :: 1300   binds_to :: g_ubnk_tab  height ::  669 .

     column bnkid>>    title :: "Banka"  left:: 100   binds_to :: bnkid input :: 0  .

   endtablectrl  .

endof frame .

display frame ubnk_grid .
```

Code Sample 2.164 *Display Frame Construct Sample 1*

In the example above (***Code Sample 2.164***), **ubnk_grid** frame user interface component is defined. Within **ubnk_grid** frame, one toolbar and one tablectrl are defined. This frame is displayed by using DISPLAY FRAME construct as it can seen from above example.

2.3.42. **REFRESH FRAME Language Construct**

The REFRESH FRAME language construct refresh frames defined with FRAME user interface component. REFRESH FRAME construct takes only one parameter that is the frame name. REFRESH FRAME construct can only be used within routines.

REFRESH FRAME construct requires the frame in question already being displayed. REFRESH FRAME only updates values of user interface components in frame that are bound to program variables. Example for REFRESH FRAME construct is given below:

2.3.42.1. **REFRESH FRAME construct usage**

```
beginof frame ubnk_grid .

   top  :: 0   left :: 0     width :: 1366  height :: 744  scale:: true on-init :: init_ubnk_grid.

   toolbar init_toolbar .

      top :: 0  left :: 0  width :: 1366 height :: 35 .
      tool save >>label :: "Kaydet"   image ::  save on-click :: on_save  .

   endof toolbar .

   tablectrl  bank_data .

     left :: 0    top :: 25    width :: 1300   binds_to :: g_ubnk_tab  height ::  669 .

     column bnkid>>    title :: "Banka"  left:: 100   binds_to :: bnkid input :: 0  .

   endtablectrl  .

endof frame .

refresh frame ubnk_grid .
```

Code Sample 2.165 *Refresh Frame Construct Sample 1*

In the example above (***Code Sample 2.165***), **ubnk_grid** frame user interface component is defined. Within **ubnk_grid** frame, one toolbar and one tablectrl are defined. This frame is refreshed by using REFRESH FRAME construct as it can seen from above example. As it is stated above, **ubnk_grid** must have to displayed before using REFRESH FRAME construct.

2.3.43. **REFRESH CURRENT FRAME Language Construct**

Frames are stacked in a frame chain by runtime during display. Whenever a new frame is displayed with DISPLAY FRAME construct, this frame becomes the top of frame chain. REFRESH CURRENT FRAME construct refreshes the frame that is at the top of frame chain.

REFRESH FRAME construct can only be used within routines. Example for REFRESH FRAME construct is given below:

2.3.43.1. **REFRESH CURRENT FRAME construct usage**

```
beginof frame ubnk_grid .

endof frame .

beginof frame ubnk_new_entry.

endof frame .

display frame ubnk_grid .
display frame ubnk_new_entry.

refresh current frame .
```

Code Sample 2.166 *Refresh Current Frame Construct Sample 1*

In the example above (***Code Sample 2.166***), **ubnk_grid** and **ubnk_new_entry** frame user interface components are defined Firstly **ubnk_grid** is displayed by using DISPLAY FRAME construct. After that **ubnk_new_entry** is displayed by using DISPLAY FRAME construct. Since displayed frames are stacked **ubnk_new_entry** frame becomes the top of this stack. REFRESH CURRENT FRAME construct refreshes the **ubnk_new_entry** frame since it is at the top of frame chain.

2.3.44. **RETURN PREVIOUS FRAME Language Construct**

The RETURN PREVIOUS FRAME language construct returns from currently displayed frame that is at the top of frame chain to previous frame that is in second position in frame chain.

RETURN PREVIOUS FRAME construct takes only one parameter that is the frame name that is at the top of frame chain. Runtime checks whether the specified frame is at the top of frame chain. RETURN PREVIOUS FRAME construct can only be used within routines. Example for RETURN PREVIOUS FRAME construct is given below:

2.3.44.1. **RETURN PREVIOUS FRAME construct usage**

```
beginof frame ubnk_grid .

endof frame .

beginof frame ubnk_new_entry.

endof frame .

display frame ubnk_grid .
display frame ubnk_new_entry.

return previous frame from ubnk_new_entry .
```

Code Sample 2.167 *Return Previous Frame Construct Sample 1*

In the example above (***Code Sample 2.167***), **ubnk_grid** and **ubnk_new_entry** frame user interface components are defined Firstly **ubnk_grid** is displayed by using DISPLAY FRAME construct. After that **ubnk_new_entry** is displayed by using DISPLAY FRAME construct. Since displayed frames are stacked **ubnk_new_entry** frame becomes the top of this stack. RETURN PREVIOUS FRAME construct returns from the **ubnk_new_entry** frame that is at the top of frame chain to **ubnk_grid** frame that is at second position in frame chain.

2.3.45. **RETURN AND REFRESH PREVIOUS FRAME Language Construct**

The RETURN AND REFRESH PREVIOUS FRAME language is very similar to RETURN PREVIOUS FRAME construct. It returns from currently displayed frame that is at the top of frame chain to previous frame that is in second position in frame chain and additionaly it refreshes the previous frame that is in second position in frame chain.

RETURN AND REFRESH PREVIOUS FRAME construct takes only one parameter that is the frame name that is at the top of frame chain. Runtime checks whether the specified frame is at the top of frame chain. RETURN AND REFRESH PREVIOUS FRAME construct can only be used within routines. Example for RETURN AND REFRESH PREVIOUS FRAME construct is given below:

2.3.45.1. **RETURN AND REFRESH PREVIOUS FRAME construct usage**

```
beginof frame ubnk_grid .

endof frame .

beginof frame ubnk_new_entry.

endof frame .

display frame ubnk_grid .
display frame ubnk_new_entry.

return and refresh previous frame from ubnk_new_entry .
```

Code Sample 2.168 *Return and Refresh Previous Frame Construct Sample 1*

In the example above (***Code Sample 2.168)***, **ubnk_grid** and **ubnk_new_entry** frame user interface components are defined Firstly **ubnk_grid** is displayed by using DISPLAY FRAME construct. After that **ubnk_new_entry** is displayed by using DISPLAY FRAME construct. Since displayed frames are stacked **ubnk_new_entry** frame becomes the top of this stack. RETURN AND REFRESH PREVIOUS FRAME construct returns from the **ubnk_new_entry** frame that is at the top of frame chain to **ubnk_grid** frame that is at second position in frame chain and additionaly refreshes the **ubnk_grid** frame.

2.3.46. **GET ROW ID FROM EVENT INFO Language Construct**

The GET ROW ID FROM EVENT INFO language allows getting row id from GUI events. VERD language runtime internally keeps row and column information in events such as click, double click or change on TABLECTRL and GRID user interface elements. VERD language runtime also keeps information about selected row of DROPDOWN user interface element.

The GET ROW ID FROM EVENT INFO only allows transferring row information into a parameter whose type is built-in **unsigned int** type or internal structure or internal structure reference field whose field type is **unsigned int**. Examples for GET ROW ID FROM EVENT INFO construct are given below.

2.3.46.1. **GET ROW ID FROM EVENT INFO construct usage with parameter**

```
data row_id type unsigned int.

get row id into row_id from event info .
```

Code Sample 2.169 *Get Row Id From Event Info Construct Sample 1*

In the example above (***Code Sample 2.169***), the **row_id** parameter, whose type is built-in **unsigned int** type is defined. Using the GET ROW ID FROM EVENT INFO construct, the row information about last event that is stored internally by runtime, is transferred to **row_id** parameter.

2.3.46.2. **GET ROW ID FROM EVENT INFO construct usage with internal structure field**

```
type cell_info >> row_id type unsigned int,
                  col_id type unsigned int .

data lc_cell_info type cell_info.

get row id into lc_cell_info-row_id from event info ..
```

Code Sample 2.170 *Get Row Id From Event Info Construct Sample 2*

In the example above (***Code Sample 2.170***), **cell_info** type and **lc_cell_info** internal structure whose type is **cell_info** are defined. Within **cell_info** type, **row_id** field whose type is built-in **unsigned int** type and **col_id** field whose type is also built-in **unsigned int** type are defined.

Using the GET ROW ID FROM EVENT INFO construct, the row information about last event that is stored internally by runtime, is transferred to **row_id** field of **lc_cell_info** internal structure.

2.3.46.3. **GET ROW ID FROM EVENT INFO construct usage with internal structure reference field**

```
type cell_info >> row_id type unsigned int,
                  col_id type unsigned int .

data lc_cell_info_ref type cell_info.

get row id into lc_cell_info_ref->row_id from event info ..
```

Code Sample 2.171 *Get Row Id From Event Info Construct Sample 3*

In the example above (***Code Sample 2.171***), **cell_info** type and **lc_cell_info_ref** internal structure reference whose type is **cell_info** are defined. Within **cell_info** type, **row_id** field whose type is built-in **unsigned int** type and **col_id** field whose type is also built-in **unsigned int** type are defined.

Using the GET ROW ID FROM EVENT INFO construct, the row information about last event that is stored internally by runtime, is transferred to **row_id** field of data that is referenced by **lc_cell_info** internal structure reference.