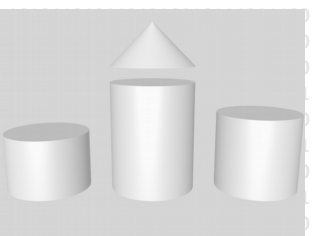


0110100100111001010110101101  
0100101110111000101110101010  
1011101100101001010110101010  
1001101011010010011100101011  
0101101010010111011100010111  
0101010101110110010100101011  
01010100110011010010011100



## ODABA NG

100110100100111001010110101  
1010100101110111000101110101  
0101011101100101001010110101  
0101001100110100100111001010  
1101011010100101110111000101  
1101010101011101100101001010  
1101010101001100110100100111  
0010101101011010100101110111  
0001011101010101011101100101  
0010101101010101001100110100  
1001110010101101011010100101  
11011100010111010101011101  
1001010010101101010101001100  
1101001001110010101101011010  
1001011101110001011101010101  
01110110010100010101101010101  
0011001101001001110010101101  
0110101001011101110001011101  
0101010111011001010001010101  
101010011001101001001110010  
1011010110101001011101110001  
0111010101010111011001010010  
10110101010011001101001001  
1100101011010110101001011101  
1100010111010101010111011001  
0100101011010101010011001101  
0010011100101011010110101001  
0111011100010111010101010111  
0110010100101011010101010011  
00110100010011100101011010110  
1010010111011100010111010101  
0101110110010100101011010101  
0100110011010010011100101011  
0101101010010111011100010111  
0101010101110110010100101011  
0101010100110011010010011100  
1010110101101010010111011001  
010111010010111011001001  
101011010010111011001001  
0101110101010101010101010101  
0101110101010101010101010101  
1101110101010101010101010101  
1100110101010101010101010101  
1010110101010101010101010101  
0101110101010101010101010101  
0101110101010101010101010101  
1101110101010101010101010101  
1101110101010101010101010101  
1101110101010101010101010101

## ODABA Script Interface - OSI

run

1101010101001100101001

Software Werkstatt



**run Software-Werkstatt GmbH**  
**Winckelmannstrasse 61**  
**12487 Berlin**

Tel: +49 (30) 609 853 44  
e-mail: [run@run-software.com](mailto:run@run-software.com)

Berlin, March 2018

# Contents

<b>1</b>	<b>Introduction.....</b>	<b>5</b>
<b>2</b>	<b>Overview.....</b>	<b>6</b>
2.1	Running OSI.....	8
2.2	Defining Data Sources.....	10
2.3	INI-file for OSI.....	11
<b>3</b>	<b>How to Write an OSI Script File.....</b>	<b>12</b>
3.1	Database References.....	13
<b>4</b>	<b>File References.....</b>	<b>14</b>
<b>5</b>	<b>Data Types.....</b>	<b>15</b>
5.1	Basic Data Types.....	16
5.2	Enumerated Data Types.....	20
5.3	Structure Definitions.....	21
5.4	Interface Definition.....	23
5.4.1	Interface Exports.....	24
5.5	Class Definitions.....	26
5.5.1	Class header.....	27
5.5.2	Type Property List for Persistent Classes.....	28
5.5.3	Class Exports.....	32
5.5.4	Class Extensions.....	35
5.6	View definition.....	36
5.6.1	View Header.....	37
5.6.2	Type Property List for Views.....	43
5.6.3	View Members.....	45
5.7	Aggregation schema.....	47
5.7.1	Aggregation example.....	48
5.7.2	Aggregation levels.....	49
5.7.3	Accessing aggregation collections.....	51
5.8	Template Data Types.....	53
5.9	Member Definitions.....	55
5.9.1	Inheritance.....	57
5.9.2	Members.....	58
5.9.3	Attributes.....	62
5.9.4	References.....	65
5.9.5	Relationships.....	70
5.9.6	Keys and Key References.....	76
<b>6</b>	<b>Variables.....</b>	<b>80</b>
6.1	Database Variables.....	85
6.2	Global Variables.....	87
6.3	Self variable and execute operator.....	88
6.4	Lookup priorities.....	89
<b>7</b>	<b>Functions.....</b>	<b>91</b>

7.1	Function Header.....	93
7.1.1	Function Options.....	94
7.1.2	Type of Returned Value.....	98
7.1.3	Function Parameters.....	99
7.2	Function Body.....	101
7.2.1	Variable Definitions.....	102
7.2.2	Processing.....	104
7.2.3	Error handling.....	105
7.2.4	Final Section.....	109
7.3	Constructor.....	110
7.4	Statements.....	111
7.5	Global Functions.....	112
7.6	Class Functions.....	113
7.7	Local Functions.....	114
<b>8</b>	<b>Operation Reference.....</b>	<b>115</b>
8.1	Syntax Functions.....	116
8.1.1	Process Flow Operations.....	117
8.1.2	Built-in Operations.....	121
8.1.3	Conditional operands.....	129
8.1.4	Query Operations.....	130
8.2	Using transient variables.....	140
8.3	Operation paths.....	141
8.4	Dynamic function calls.....	147
8.5	Built-in Class Functions.....	148
<b>9</b>	<b>OSI Templates.....</b>	<b>150</b>
9.1	ASCII templates.....	152
9.2	HTML Templates.....	155
9.3	Template specifications.....	157
9.4	Template Result.....	160
9.5	Debug templates.....	162
<b>10</b>	<b>Trace function calls.....</b>	<b>163</b>
<b>11</b>	<b>OSI-Debugger.....</b>	<b>165</b>
11.1	Breakpoints.....	167
11.2	Reload OSI functions.....	170
11.3	Debug Commands.....	171
11.4	Debug functions.....	175
<b>12</b>	<b>Running OSI under OShell.....</b>	<b>176</b>
<b>13</b>	<b>References.....</b>	<b>177</b>

# 1 Introduction

ODABA is an object-oriented database system that allows storing objects and methods as well as causalities. As an object-oriented database, ODABA supports complex objects (user-defined data types), which are built on application relevant concepts.

ODABA-applications are characterised by a high flexibility that is achieved by supporting in addition to object (concept) hierarchy, multifarious relations between objects (master and detail relations, relations between independent objects and others). This way conditions and behaviour of objects in the real world can be represented considerably better than in relational systems.

ODABA-applications cannot only be drawn up as event-driven applications within the field of the graphical surface but also at the database level. This is one more way in which the application design is very close to the problem.

This makes ODABA-applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

## **Platforms**

ODABA supports windows platforms (Windows95/98/Me, Windows NT and Windows 2000) as well as UNIX platforms (Linux, Solaris).

One can build local applications or client server applications with a network of servers and clients.

## **Interfaces**

ODABA supports several technical interfaces:

- C++, COM as application program interface (this allows e.g. using ODABA in VB scripts and applications)
- ODBC (for data exchange with relational databases)
- XML (as document interface as well as for data exchange)

## **User Interfaces**

ODABA provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA provides a special ODABA GUI builder.

## 2 Overview

The ODABA Script Interface (OSI) is a script language for accessing and updating ODABA databases. The OSI script language included the object definition language (ODL) and a method language for implementing OSI functions (ODABA Script Language - OSL). OSL and ODL together form the OSI language. The OSL syntax is similar to JAVA and easy to understand for C++ or JAVA programmers. On the other hand, the OSL syntax follows the suggestions of the ODMG database management group as defined in the "Object Data Standard: ODMG 2003".

To be JAVA and ODMG compliant and to support several ODABA specific features, the OSI language provides different syntax variants in several places. While programmers prefer JAVA like syntax, we use JAVA syntax as far as possible for OSI function examples. For model definitions (ODL examples), we try to be as close as possible to the ODMG syntax, but often we have to introduce ODABA specific extensions (options and qualifiers), which are not part of the ODMG language.

### Features

	OSI provides most features an object-oriented programming language is supposed to support. Besides, it supports embedded OQL language elements (queries) and C++ program interfaces.
Run-time classes	OSI allows defining classical queries (SELECT statements) but also defining procedures or complete applications. Besides resources (classes, structures, functions etc.) defined in the project resource database, OSI allows defining run-time classes within the OSI script. Run-time classes are classes relevant for the application only, which will extent the definitions in the resource database.
Interfaces	From within OSI one may call functions defined in the resource database. Functions can be called as class functions via the OSI C++ interface but also as context functions. Support for calling COM functions is planned for OSI 2.0.
Functionality	OSI supports most of the ODABA access class functions for <code>odaba::Dictionary</code> , <code>odaba::Fatabase</code> , <code>odaba::ObjectSpace</code> and <code>odaba::Property</code> . This includes browsing functions as well as update functions.
Built-In functions	Several built-in classes and functions are provided to support file output and several system features. One may extent the built-in functionality by providing a libraries for your support classes.

## C++

Even though the OSI syntax is rather similar to C++, there are some important differences, which result from the fact, that OSI operates on a database and not only on memory. Syntactically, OSI does not make differences between transient (application) and persistent (database) instances.

```
person.age = 25;
```

The statement above could apply to a database `Person` instance, but also to a `Person` variable defined in the application.

### Restrictions

OSI is syntactically similar to C++, but does not support a number of C++ features (e.g. operator overloading, or local classes and types).

For using and accessing standard libraries from within OSI functions, you need to define an OSI interface to those classes. OSI provides a simple interface mechanism, which allows defining an interface to any C++ class.

OSI 1.0 ignores access privileges for complex data type members (private, protected, public). They can be specified but will be ignored. Support for access privileges is planned for OSI 2.0

### Differences

OSI does not support pointer variables, but only “by reference” and “by value” variables. The variable semantics differs slightly from the C++ variable semantics, since OSI variables got a cursor functionality (see “Variables”).

Different is also the handling of virtual methods, since OSI considers a method as virtual only for the class, where the method is defined as virtual, i.e. the `VIRTUAL` property for methods is not inherited in OSI. Functions declared as virtual are overloaded, when a specialized class defines an appropriate overload.

OSI allows defining functions with the same name for one class. In contrast to C++, however, OSI provides implicit data conversion features also for parameters. Hence, OSI cannot select the proper function by parameter types. Instead, OSI 1.0 looks for the number of parameters. For OSI 2.0 a “best fit” strategy is planned for selecting functions with identical names in a class.

## 2.1 Running OSI

One may run an OSI application from a command line console or from within OShell, when you have installed at least ODABA 9.0. OSI is platform independent and can be used on Windows, LINUX and UNIX platforms.

OSI requires a project resource database (dictionary), which defines the data and functional model for the application. You may also run an OSI application without dictionary as long as you are not going to store persistent data in a database.

**Usage** You may run OSI from a command line in DOS or UNIX.

```
OSI      script_file  
          [ -I:ini_file | -D:dict_path ]  
          [ -P:parameters ]  
          [ -E:entry_point ]  
          [ -DB ]
```

Besides the script file, OSI requires at least a dictionary, which can be defined in the script file, in the ini-file or as parameter.

*script\_file* The script file parameter refers to a location where the ODABA Script file is stored. When not passing an ini-file or dictionary path, the dictionary location must be defined in the script file.

*dict\_path* The dictionary path provides the location for the dictionary. It becomes necessary, when no dictionary is defined in the script file. When passing the dictionary location, the dictionary location in the script file will be ignored.

*ini\_file* Instead of a dictionary path, an ini-file can be passed. When the ini-file refers to a dictionary, this will replace the dictionary location defined in the script file. When containing a database reference, the data base location in the ini-file is replaced as well.

Besides database and dictionary, the ini-file may contain definitions of system variables, which can be referenced in the script.

*parameters* Any number of parameters separated by comma can be passed to the script file. When the list contains spaces, the option must be enclosed in “”:

```
“-P:parm1, message text, parm3”
```

Parameters are passed in the same sequence to the function referenced as entry point.



*entry\_point*

The entry point is the name of the function that is called as starting function. When no entry point is defined, "main" is assumed.



Entry point names are case sensitive, i.e. when not defining an entry point, the script file must contain a function with the name 'main'.

Debug mode

One may run OSI in debug mode by setting the debug option `-DB` when calling OSI. One may also set the debug option in the ini-file (`OSI.DEBUG=YES`) or in a system environment variable.

More details about the debug mode One may find in chapter "**OSI-Debugger**".

**Samples**

You will find sample scripts and ini-files in the ... ODABA/Sample installation folder.

## 2.2 Defining Data Sources

OSI requires a database defined in a data source. The data source definition includes at least a dictionary, but usually it consists of dictionary definition and database path.

There are different ways of providing data source definitions. Typically, the data source is defined in the script file or in an ini-file passed to OSI. In both cases, there are two ways to refer to a data source. One way is to define the data source implicitly by defining dictionary and database in the script or in the OSI section of the ini file.

The other way is defining the data source explicitly in a separate section of the ini file. In this case, the data source is defined in a section that is preceded by the data source name:

```
[DataSource1]           Data source name
```

Now, the data source can be referred to by its data source name as:

```
DATA_SOURCE=DataSource1
```

This way, it is also possible to refer to data sources defined in the database catalogue. When referring to a database catalogue, the ini-file must contain a catalogue section that defines the location of the data catalogue:

```
[DATA-CATALOGUE]
```

Data catalogues can be provided locally and on the server side. In one application, however, one may refer to only one data catalogue. How to define data sources and file locations in the data and file catalogue one may find in the “ODABA – Server” documentation.

### **Multiple data sources**

Functions within an OSI script may refer to multiple data sources. When referring to more than one data source, the ini-file should contain definitions for all required data sources.

The OSI script may refer directly to dictionary and database paths. In this case, the script becomes dependent on the data location, which can be avoided by referring to data sources in an ini-file.

## 2.3 INI-file for OSI

An ini-file defines the data source, input and output files and other process specific parameters. The following example refers to the specification of the sample database source based on an ODABA database.

```
[SYSTEM]                                system section
DICTIONARY= C:\ODABA\ODE.SYS
```

```
[OSI]                                    OSI section
DICTIONARY=C:\ODABA\ Sample\Sample.rot
DATABASE= C:\ODABA\Sample\Sample.dat
NET=YES
COLLECTION=Company.employee
DEBUG=YES
```

<b>SYSTEM</b>	The system section allows providing a system dictionary, which mainly contains error messages and descriptions.
DICTIONARY	The dictionary refers to the system database (usually ode.sys), which contains error messages and message descriptions for system (database) errors.
<b>OSI</b>	The OSI section defines data source and runtime options for running OSI script files.
DICTIONARY	The dictionary refers to the application database (resource database), which contains data model definitions, OSI scripts and other run-time resources.
DATABASE	The database option contains the path to the database file.
NET	Indicates, whether the database allowed shared access (YES) or not (NO).
COLLECTION	Allows defining an access path to a collection to be processed by the OSI function (optional).
DEBUG	In order to debug OSI functions, this option may be set to true. The option may be set also by the -DB option when calling OSI or OShell.  When running GUI applications in debug mode (debug=true), those must be called via <b>code (code.exe)</b> and following option has to be set in addition:  CONSOLE_APPLICATION=YES  in order to redirect application output to console.

## 3 How to Write an OSI Script File

An OSI script file includes database locations, class and view definitions, definitions of global variables and functions. A script file must have at least one function, which is declared as entry point. By default, this is a function with the name "main".

Script files for complex applications may include other script files, e.g. one script file per class.

**Class extensions** Within a script file, one may extend the functionality of a class defined in the resource database, but you cannot extent the structure.

**Syntax** The syntax of an OSI script file is compliant with the ODABA ODL (object model definition language). Details are described in the "ODABA Language Reference".

**Keywords** The OSI language accepts most keywords in lower case letters and capital letters (but not mixed). For better readability, we use capital letters for keywords in the examples. All capital letter keywords are reserved name and cannot be used as identifiers (e.g. variable names). Most lower case keywords can be used as keywords but also as variable names. This might cause syntax ambiguities in some cases and therefore, we suggest using capital letter keywords, always.

**Comments** Comments can be written at the line end starting with //. After a comment no code can be defined on a line. Comments may also start on first line position.

```
// location for resource database  
DICTIONARY=c:/ODABA/Sample/Sample.res; // sample resources
```

Comment blocks are comments enclosed in /\* ... \*/. Comment blocks can be defined at specific places, only. One may define comment blocks before class and view definitions, function definitions, file references or variable definitions. We suggest, however, using comments rather than comment blocks.

```
/* This file includes the class declarations  
   Person and its extent definitions          */  
INCLUDE c:/ODABA/Sample/Person.osi;
```

## 3.1 Database References

Database references include the definition of locations for the resource database (dictionary) and for the database containing the application data.

Database references must be defined at the beginning of the script file and cannot be defined in included script files.

```
DICTIONARY = c:\ODABA\sample\sample.res;  
DATABASE = c:\ODABA\sample\sample.res;
```

Another way of defining a database is referring to a predefined data source:

```
DATASOURCE = Sample;
```

Data sources must be defined in the data catalogue, which is defined in the OSI ini-file in the [DATA\_CATALOGUE] section or in a separate section of the OSI ini-file with the data source name.

### Reference priorities

Database references can be defined in different places, but at least in one place, the databases must be defined. The database definition in the script file can be overwritten by the database definition in [OSI] section of the ini-file passed to the OSI program call, i.e. the database references in the script file have lowest priority.

Highest priority have database references in the program call (dictionary parameter), which will overwrite database references in the ini-file as well.

## 4 File References

File references are used to include specifications defined in other script files. This allows reusing definitions for different applications.

File references can be defined at any place outside a class or view definition, outside a function definition and outside comment blocks.

```
INCLUDE c:/ODABA/Sample/Person.osi;
```

One cannot use file references to include variable definitions in a function or in a class definition. Thus, each included file must provide complete definitions in the sense of the OSI syntax. This means, each included file must contain any number of valid class, view, variable or function definitions.

Included script files may contain file references, again (as in the Person.osi example:

```
INCLUDE c:/ODABA/Sample/PersonStruct.osi;  
INCLUDE c:/ODABA/Sample/PersonExpr.osi;
```

### Splitting definitions

Class and view definitions can be defined in several blocks. Thus, one section may contain the structure definitions while the other contains the function definitions. Each class or view definition section must begin the CLASS or VIEW keyword and referring to the same class name.

Specifications of different definition sections for a class or view are collected for the corresponding class or view, i.e. each definition section will extend the class or view definition.

Included files may contain a debug information path (seeOSI-Debugger), which refers to debug commands for the included file. Included files may also contain dictionary and database information, but this will be ignored.

# 5 Data Types

Within an OSI script one may refer to basic data types, user-defined data types and template types.

User-defined types are supported as complex data types (structure, class) and enumerated data types.

## Type hierarchy

Most programming environments more or less support the following type hierarchy

```
Data type
  basic data type
    text type
    numerical type
    date/time type
    Boolean type
  user-defined data type
    enumerated data type
      typed enumeration
    complex data type
      structure
      class
      interface
        view
    type definition
    union type
  template type
    collection
  constant
```

OSI does not make a difference between an interface definition and a class, i.e. an interface definition is considered as class definition, which does not support persistence. OSI supports interface definitions, but those are handled like class definitions.

OSI supports two extensions in the type hierarchy. One are typed enumerations, which define a link between category and class. The other are views, which are considered as method and interface in OSI.

OSI 1.0 does not support union types. OSI 1.0 does also not support the declaration of constant values.

## 5.1 Basic Data Types

Basic data types provide a number of built-in data types. For convenience, basic data types have ODABA names, but also synonyms, which correspond to Java/C++ and ODMG data types.

Most basic data types in OSI do have optional size parameters.

CHAR(40)	name;	// different from: char	name[40];
INT(10)	count;	// same as: long	count;

The size parameter corresponds to the number of relevant characters or digits of the data type. The size parameter differs from the dimension value, which can be accessed by index.

### Text types

Text types are provided for storing text with different properties.

**CHAR** defines a text buffer, which is padded with blanks. The **CHAR** type differs from the JAVA/C++ specifications, which have always size 1.

The type is obsolete and **STRING** should be used, instead. Operating with **CHAR** values will truncate trailing spaces automatically.

Synonyms: `char`

Default size: 40

**CCHAR** Coded character fields are used for storing text data in a coded form in the database to avoid that sensitive information can be read directly in the database. There is no correspondence in C++/Java or ODMG standard.

Synonyms: `cchar`

Default size: 40

**STRING** Strings are 0-terminated text strings terminated with 0 after the last valid character. The size parameter for a string value defines the maximum number of relevant characters not including the terminating 0. **STRING** types correspond to the string template type in ODMG (`string<size>`)

Synonyms: `string`, `MEMO`, `string<size>`

Default size: automatic



UTF8 defines a 0-terminated Unicode text string, which is terminated with double 0 after the last valid character. The size parameter for a Unicode string defines the maximum number of relevant characters not including the terminating 0. There is no correspondence in C++/Java or ODMG standard.

Synonyms: `utf8`

Default size: automatic

## Numeric types

Numeric types are used for presenting and storing numeric values. Numeric types are supported as integer types and float number types.

INT describes an integer value. Two size parameters can be passed to integer types, defining the maximum number of digits, which can be stored in the field, and the precision.

The precision value defines the number of positions behind the decimal point.

```
// number with 7 digits before and 2 after the decimal point:
INT(9,2)    income;
// 10 digits national income measured in 1000 €:
INT(10,-3)  national_inc;
```

The precision value allows defining a decimal point position as well as a factor. When data conversion becomes necessary, it takes into account the precision of source and target value.

Synonyms: `int`, `DEC`  
`short` - `(INT(4))`  
`long` - `INT(10)`  
`long long` - `INT(19)`

Default size: 10

Default precision: 0

BIT is an integer type with bit boundaries instead of byte boundaries. The type is supported for compatibility reasons, only. BIT types are supported for transient values and cannot be stored in the database.

Synonyms: `bit`

REAL is an 8 byte floating value (double). Floating values do not have size and precision parameters.

Synonyms: `double`, `real`

FLOAT describes a 4 byte floating value without size and precision.

Synonyms: `float`

## Time types

Time types provide data for date and time points or durations.

TIME Time values are provided as time measured in 1/100 seconds. Practically, a TIME value is an INT(10) value, but it provides a number of features for displaying time correctly, e.g. when converting TIME to STRING.

The default string format for TIME values is: `hh:mm:ss,cc`.

Synonyms: `time`

DATE Date values are stored as number of days since 1.1.1870. Practically, a DATE value is an INT(10) value, but it provides a number of features for displaying dates correctly, e.g. when converting DATE to STRING.

The default string format for DATE values is: `yyyy/mm/dd`.

Synonyms: `date`

DATETIME is a timestamp value consisting of DATE and TIME value.

The default string format for DATETIME values is: `yyyy/mm/dd|hh:mm:ss,cc`.

Synonyms: `datetime`, `timestamp`

## Other data types

In addition to text, numerical and time types, OSI supports some special data types as described below.

BOOL Is a specific data type for storing Boolean values. Typically, Boolean types appear as result of Boolean expressions. Boolean values may have values `true` and `false` (YES and NO), only.

Synonyms: `BOOL`, `LOGICAL`, `boolean`, `logical`

ANY defines data of any (or unknown) type. Variables of ANY data type can be passed (or defined) by reference, only, and never by value. Defining ANY type as direct value or by value indicates a non-existing value (e.g. a function or expression returning ANY or VOID does not return any value).

```
VOID    SetDate();           // does not return a value
VOID    & GetInstance();     // returns an instance of any type
```

BINARY

allows storing binary large objects. The size for binary large objects is dynamically adjusted when storing data for binary large objects.

Synonyms: BLOB, binary

## 5.2 Enumerated Data Types

Enumerated data types are enumerated value domains defined by the user. A value domain consists of a number of names (string values, which might be associated explicitly with a numerical value. Regardless of the definition of numerical values each enumerated value gets a numerical value, which is stored instead of the string value as data in the database or in variables.

```
ENUM Sex {
    male = 1,
    female = 2,
    undefined = 0
};
```

### Type-based enumerations

Type-based enumerations can be defined, when the categories in the enumeration apply to a set of objects of a given type. Often, in such a situation enumerators become so-called discriminators, which associate a specialized type with each category.

Considering Sex as a person property and supposing that specialized types Man and Woman are defined inheriting from Person, the following type-based enumeration could be defined:

```
ENUM Sex (Person)
{
    male(Men) = 1,
    female(Woman) = 2,
    undefined(Person) = 0
};
```

Defined enumerations can be used as data type for local and global variables, structure members and class attributes.

## 5.3 Structure Definitions

Structure definitions are used for defining complex data types, which are instantiated as transient instances or within other complex data types (structure, class, ...).

In contrast to class definitions, structure definitions do not support inheritance, type properties, relationships and references. Structures only contain members, which are attributes.

Structure definitions are introduced by the `STRUCT` keyword followed by the structure name. A list of structure `members` defines the attributes of the structure.

```
STRUCT str_name
{
    member (*)
};
```

In contrast to C++, structure definitions do not support declarators, i.e. structure definitions and declarations are clearly distinguished.

Structures are typically used for storing transient data or as type for attributes.

```
STRUCT Address
{
    STRING(6)    zip;
    STRING(40)   city;
    STRING(80)   street;
    STRING(6)    number;
};
```

### Structure Member

A structure may contain members of basic or complex data type (user-defined types). Within a structure, members can be referenced directly (imbedded) or by reference.

#### By reference members

When defining a structure member by reference, the instance is not stored within the structure instance. So-called “by reference” members behave like pointers, i.e. when assigning a value to such a member, only the pointer to the passed data is stored in the member.

As long as no value is assigned, no data is available for the data, i.e. accessing the member will cause an error.

```
STRUCT Address
```

```
{  
    STRING(6)    zip;  
    City        &city;  
    STRING(80)   street;  
    STRING(6)    number;  
};
```

In the example above, city has been replaced by a complex data type City, which is referenced in the city member.

More details about member definitions are described in chapter “Members”.

## 5.4 Interface Definition

Interface definitions provide an intensional definition of a complex data type. An interface definition can be extended to the more specific class definition, which contains extensional specifications in addition.

The main difference between an interface and a class is, that an interface cannot define persistence, i.e. interface definitions are considered as complex data types for transient data. Thus, one cannot define extents, keys and sort orders for an interface, which are specific properties of persistent classes.

```
INTERFACE iname [ inheritance_spec ]
{
  [ exports(*) ]
};
```

Many of the features defined for the interface members (exports) are describing extensional features, which are not of interest in the context of an interface definition, but OSI will accept them. Practically, OSI does not make any difference between `INTERFACE` and `CLASS` definitions, but for compatibility and conceptual reasons, it is suggested to clearly distinct `INTERFACE` and `CLASS` definitions.

In this chapter, only the member features relevant for the interface are described. Extended member definitions for class members are described in chapter "Member Definition".

### inheritance

Inheritance for an interface definition describes the intensional specialization of a complex data type. Extensional features provided for inheritance member definitions should not be used, when defining interfaces.

```
Interface IPerson : PUBLIC Person person
```

The only qualifier supported for an interface inheritance specification is `INVERSE`, which allows defining a reference from a class instance to its interface.

```
Interface IPerson : PUBLIC Person person INVERSE p_interface
```

## 5.4.1 Interface Exports

The exports of an interface according to ODMG includes the following features:

- Type declarations (`type_dcl`)
- Constant declarations (`const_dcl`)
- **Attribute declarations** (`attr_dcl`)
- **Reference declarations** (`ref_dcl`)
- **Relationship declarations** (`rel_dcl`)
- Exception declarations (`except_dcl`)
- **Method definitions** (`method`)

OSI does not support type and constant declarations in an interface definition, i.e. those can be defined but will be ignored.

Also exception declarations are not supported in OSI 1.0, but are planned to be supported in future versions.

The interface definition supports all types of member definitions. Features, however, related to key definitions or extents (super set relations) should not be used in an interface definition.

```
INTERFACE IPerson : PRIVATE Person
{
  ATTRIBUTE {
    INT(3)      age = (Date() - birth_date).year;
  };
  REFERENCE Address current_location;

  RELATIONSHIP {
    Person                receiver[0]
                        INVERSE sender;
    Person SECONDARY     sender[0]
                        INVERSE receiver;
  };
  ... Method definitions
};
```

### Attribute

Attribute definitions are similar to structure members. An interface containing only attribute definitions, could also be defined as structure.

Details for the attribute definition are described in "Attributes.



Initializing attributes	Initial values for interface attributes are set, when an interface object is constructed. The type of expressions allowed for initializing depends on the programming language that implements the interface. Usually, constants are supported.
<b>Reference</b>	Interfaces do not define references, but use pointers or generic type attributes for implementing references.
<b>Relationship</b>	Interfaces allow the definition of relationships, but it depends on the implementation language for the interface, how far relationships are supported.  Relationship definitions in interfaces are restricted to options and qualifiers, which make sense for transient data types.
Type options	None of the defined type options for relationships are supported for interfaces.
Collection Options	Collection options are not supported in interface definitions and will be ignored.
Data types	As domain type or data type for relationships in an interface definition, only interface types are allowed.
Base collections	Base collections are not supported for relationships in interface definitions.
Order keys	Order keys are not supported for relationships in interface definitions.
<b>Methods</b>	For loading an interface definition, all method types defined in ODABA can be referred to. When running OSI, however, OSI functions are supported, only.  The details for defining an OSI function are described in "Functions".

## 5.5 Class Definitions

Class definitions in an OSI script file contain the structure definition and a number of function definitions. One may define transient class definitions in OSI function, which are known to the application, only, but one may also extent class definitions defined already in the resource database.

A class definition consists of a class header, class properties and a class definition. The general format for a class definition follows the rules defined below:

```
CLASS classname [guid] [ inheritance_spec | extends_spec ]
      [ type_property_list ]
{
  [ exports(*) ]
};
```

## 5.5.1 Class header

Besides the class name the class header contains the inheritance specification. The general format of the class header is:

```
CLASS classname [guid] [ inheritance_spec | extends_spec ]
```

Usually, class headers are simply defined as e.g.:

```
CLASS Person
```

**guid**

The GUID option indicates that global unique identifiers (GUID) are to be created for each instance of the class.

GUIDs can be created for classes, only, which inherit from `__OBJECT`. Otherwise, the option is ignored.

**inheritance**

Since OSI does not differ between interface and class, the OSI class header supports both, the EXTENTS specification for classes as well as the inheritance specification for interfaces, i.e. one may use the inheritance operator `:` as well as the EXTENDS keyword followed by one or more base type references.

```
CLASS Employee : PUBLIC Person person BASED_ON Persons  
                    INVERSE pers_ref
```

In contrast to other programming languages, base types in OSI get a name (person in the example above), which allows distinguishing between different bases types of the same type.

Moreover, ODABA considers base types as class members (specialization of relationships), without losing the advantages of inheritance. For compatibility reasons, however, base types are defined as inheritance rather than as class members.

Other extensions are the definition of base collections, inverse references and several additional options, which result from the concept of shared base instances.

When inheriting from more than one base type, base type references are separated by comma.

## 5.5.2 Type Property List for Persistent Classes

Type property lists are important for persistent classes, i.e. for classes, which will store instances in the database. Application classes usually do not have a type property list.

**Persistent classes** For test purposes or when creating view classes, it might be useful in some case defining persistent classes in an OSI application, as well.

OSI will create appropriate extent definitions and can store data for these classes in a temporary database. This temporary database, is deleted automatically when terminating the application.

Thus, defining a persistent class in an OSI application does not really create persistent instances for the class, which are accessible after terminating the application.

**Property definitions** Type or class properties are defined as key and extent definitions and alignment properties.

```
CLASS Person
(
  KEY {
    IDENT_KEY ik_pid(pid);
    sk_Name (name,first_name);
  };
  EXTENT Persons MULTIPLE_KEY OWNER
    ORDERED_BY (ik_pid UNIQUE SUPPRESS_EMPTY, sk_Name);
)
```

**Key definitions** Key definitions allow defining one or more keys, which can be used later on for creating indexes for collections to provide fast access.

One of the keys can be marked as identifying key, which usually determines the default order when accessing data in a collection.

Details for key definitions and how to use keys are described in “Keys and key references”.

**Extent definitions** Usually, one extent is defined for a persistent class, but one may define also a number of extents inclusive specific set relations (intersect, union etc.). Each extent must have at least one order key (index).

For an extent, several collection options can be defined, which are explained below.

**Collection options**    Collection options (`rel_option`, `ref_option`, `extent_option`) can be set to enable specific features for an ODABA collection. Collection options for extents include many collection options for references and relationships.

**GUID**    The `GUID` option indicates that global unique identifiers (GUID) are to be created for each instance, which is created for the extent. This option need to be set only, when the referenced data type did not define the `GUID` option.

GUIDs can be created for classes, only, which inherit from `__OBJECT`. Otherwise, the option is ignored.

```
EXTENT Persons GUID ORDERED_BY (ik_pid);
```

**WEAK\_TYPED**    An extent may refer to instances of different types, which are typically specializations of a common base type. For weak-typed extents, the data type for the extent defines the common base type for the instances in the collection.

```
EXTENT Persons WEAK_TYPED ORDERED_BY (ik_pid);
```

This extent may contain persons, but also women and men, when `Woman` and `Man` have been defined as specialized classes of `Person`.

For creating an instance of a given type, the type must be set for the extent in advance (`setType`).

```
Persons.setType('Woman');  
Persons.insert('ID0033');            // creates a Woman instance
```

**DELETE\_EMPTY**    This option is ignored for extents.

**MULTIPLE\_KEY**    Extents are always considered as multiple indexed collections and this option need not to be set but can be defined.

```
EXTENT Persons MULTIPLE_KEY ORDERED_BY (ik_pid);
```

**UPDATE**    Extents are considered as resources being used by different applications, i.e. this option is set by default.

**OWNER**    An extent can be defined as the owner of the instances referring to. When removing an instance from an owning extent, the instance will be automatically deleted.

```
EXTENT Persons OWNER ORDERED_BY (ik_pid);
```

In an ODABA database, each instance belongs to exactly one owning collection. Thus, not owning extents must always have a superset, which owns the instances.

```
EXTENT Women ORDERED_BY (ik_pid) SUPERSET Persons;
```

Considering extents `Woman` and `Men` as subsets of `persons`, which is the owning collection, `Persons` becomes the superset for the `Women` extent.

`NO_CREATE`

This option prevents instances from being created in an extent. When the option is set, only existing instances can be added to the extent. The option requires a superset, always.

This option should never be set for owning relationships, because one cannot create instances at all in this case.

```
EXTENT Women NO_CREATE ORDERED_BY (ik_pid)
SUPERSET Persons;
```

Defining the `Women` extent as `NO_CREATE`, one may add persons to the `Women` extent, which already exist in the `Persons` extent.

`DEPENDENT`

This option is ignored for extent collections.

`SHARED`

Shared collections are collections, which can be shared between different transactions. This requires specific key locking strategies to avoid assigning duplicate keys to unique indexes. For extents, this option should always be set.

```
EXTENT Persons SHARED ORDERED_BY (ik_pid);
```

`ORDERED_BY`

At least one unique order key must be defined for each extent. The order key must be a key defined in the structure the extent is based on.

More details about order key definitions are described in “Keys and key references”.

`SUPERSET`

Subset-super set relations are bi-directional and need to be defined only in one direction. As long as the set relations are consistent, one may define also both directions.

One or more extents can be defined as super sets for an extent.

```
EXTENT Women ORDERED_BY (ik_pid) SUPERSET Persons;
```

When defining more than one superset, the subset can be defined as intersection of its supersets.

```
EXTENT Engineers ORDERED_BY (ik_pid) SUPERSET Persons;
EXTENT Employees ORDERED_BY (ik_pid) SUPERSET Persons;
EXTENT EmployedEngineers ORDERED_BY (ik_pid)
SUPERSET INTERSECT Engineers, Employees;
```

In this example, the set of employed engineers is defined as the intersection of `Employees` and `Engineers`.

Intersect hierarchies are maintained by the database system, adding an instance to a subset, it is added automatically to all supersets (which does not require the intersect option). When deleting an instance from one of the supersets, it is deleted automatically from the subset (regardless of the intersect option, as well). But when adding an instance to one of the supersets, which already exists in the other superset(s), the instance will be added automatically to the intersect extent.

SUBSET

Subset-super set relations are bi-directional and need to be defined only in one direction. As long as the set relations are consistent, one may define also both directions.

One or more extents can be defined as subsets for an extent.

```
EXTENT Persons ORDERED_BY (ik_pid)
SUBSET Men, Woman;
```

With the subset options extended set relations as union and distinct can be defined, which are maintained by the database system:

```
EXTENT Persons ORDERED_BY (ik_pid)
SUBSET UNION DISTINCT Men, Woman;
```

The definition defines `Persons` as union of `Men` and `Women`, which are distinct to each other. In this case, you cannot create a new `Person` in the `Persons` extent, since the database system cannot decide, to which subset the new `Person` should be added. Thus, the `UNION` option implies that instances can be created for the subsets, only. The `DISTINCT` option prevents the application from adding the same `Person` to the `Men` and `Women` extent. When adding a `Person` to the `Women` extent, which already exists in the `Men` extent, it will be removed from the `Men` extent before being added to the `Women` extent.

### 5.5.3 Class Exports

Class exports are the elements defining a class. There are different categories of class exports, that can be defined for a class:

- (Inheritance/extends)
- Constants
- Types
- Exceptions
- Attributes
- References
- Relationships
- (Keys)
- Methods
  - (C++ Functions)
  - OSI Functions
  - (Templates)
  - (Forms)

#### **OSI specific**

The current version of OSI/ODL accepts constants, exception and type declarations, but does not support those. When such declarations are defined in an ODL or OSI file, they will be ignored.

When running an OSI script file, OSI functions are the only type of methods, which is supported. When loading an ODL schema, methods are considered by default as C++ functions. Other method types must be declared explicitly.

Base types and keys are not defined as class members but can be accessed as class members. Those are defined in the class header and in the type property list for the class.

Support for templates and forms is planned for OSI version 2.0. In OSI 1.0 one may already refer to forms stored in the resource database (dictionary).

Member type keywords (attribute, reference, relationship) can be placed in front of each member definition or once for a block of class members.



```

CLASS Person
{
    ATTRIBUTE {
        CHAR          pid[16];
NOT_EMPTY STRING(40) name;
        STRING(40) first_name;
        DATE          birth_date;
        Sex           sex;
TRANSIENT INT(3)    age = (Date - birth_date).year;
    };
    REFERENCE Address location;
    REFERENCE STRING(4000) notes;

    RELATIONSHIP {
        Person          children[0]
                        BASED_ON Persons
                        ORDERED_BY (sk_name UNIQUE)
                        INVERSE parents;
        Person SECONDARY parents[2]
                        BASED_ON Persons
                        ORDERED_BY (sk_name UNIQUE)
                        INVERSE children;
        Company         company
                        BASED_ON Companies
                        ORDERED_BY (ik_name UNIQUE)
                        INVERSE employees;
        Car NO_CREATE  used_cars[2]
                        BASED_ON company.cars
                        ORDERED_BY (ik_cid UNIQUE)
                        INVERSE users;
    };
};

```

### **Attribute**

Attribute definitions are similar to structure members. A class containing only attribute definitions, could also be defined as structure.

### *Initializing attributes*

For initializing attributes an initial value or an expression can be assigned to the attribute. The assigned value is computed and set, when a new object instance is created or when a persistent instance is read.

### *Constraint*

Constraints allow defining rules for validity checks on the attribute. Persistent instances, which violate the constraint, are not stored to the database.

When trying to assign a value in an expression, that violates a constraint, an exception is raised.

**Reference**

References in a class definition allow defining details for an object instance. References behave like collection, but do not support inverse references as relationships do.

References can be defined as persistent or transient references. Only transient references may have an initial value.

Details about reference definitions are described in “References”.

**Relationship**

Relationships in a class definition allow defining links to other (independent) object instances. Relationships behave similar to references, but support a number of additional features.

Relationships in a class definition cannot be defined as transient, i.e. relationships are always considered as having the same persistence state as the class instance has.

Details about reference definitions are described in “Relationships”.

## 5.5.4 Class Extensions

A class definition in an OSI script file consists of the data definition and a number of method definitions (OSI functions, C++ functions etc). Since an OSL script supports OSI functions, only, we will not consider C++ functions here.

One may define transient class definitions in your function, which are known to the application, only, but one may also extent class definitions defined already in the resource database.

When extending class definitions defined in the resource database, the class extension may contain only functions. Class extensions for transient (application) classes may contain properties as well as functions.

## 5.6 View definition

OSI supports two ways of defining views, as method (similar to traditional queries) or as complex data type. Here, the view definition as complex data type is described. How to define a view as method is described in section “OSI Function definitions”.

As complex data type, a view can be referenced in other views as complex data types. One may also store view instances to the database making the view persistent.

```
VIEW vname [from_spec] [where_spec] [group_spec] [having_spec]
           [type_property_list]
{
  [ view_member(*) ]
};
```

The view header defines the rules of providing view instances from a number of data sources. The `view_member` definitions define the view elements, i.e. the members the view consists of. The view elements correspond to the field definitions in the `SELECT` clause of a traditional view.

A view may be defined as local view (class method) or as global view. Local views refer to class members as data source while global views refer to extents.

In order to instantiate a view, one may define extents in the database model (global view) or view definition. For local views, one may also refer via transient properties to view definition by referring to the view as data type.

One may also call the view definition like a method in program by means of the `SELECT` clause referring to a pre-defined view structure..

## 5.6.1 View Header

Besides the view name the view header defines the view source in terms of source definitions, selection criteria and grouping specifications.

```
VIEW vname [from_spec] [where_spec] [group_spec] [having_spec]
           [order_spec] [type_property_list]
```

A view data type differs from a view expression, since it defines a method rather than a collection of instances. For local view definitions, the view header does not refer to specific source data collections but refers to instance properties of the owning data type definition in the `FROM` type specification. Global view definitions usually refer to extents or extent paths.

For local view definitions, the view definition may consist simply of a list of view members, in which case the view definition simply provides a number of additional attributes (or collection properties) derived from source instance properties.

### Data source

The data source for a view definition consists of one or more complex collection definitions (extent, class property, property path), which define the source for the view.

```
VIEW Retired
     FROM ( Person ) ...
```

#### Data source definition

The `FROM` expression defines the base for a select statement or view definition. The operands in the parameter list should result in collections and are passed as collection parameters to the `FROM` method.

In contrast to traditional view definitions the `FROM` clause requires parenthesis around the complex data type list. As long as the view refers to a simple data source, one may omit the `FROM` clause when defining local views (open view definition).

```
FROM ( Person, Company )
```

**Multiple sources** When defining several operands in the operand list, data types for each operand become base types for the `FROM` data type results in an instance type containing `Person` and `Company` as base types). Base types may be referred to by base type member name, which by default is the collection name passed in the `FROM` operand list.

By preceding the collection references with names, one may explicitly assign base type member names.

```
FROM ( pers = Person, comp = Company )
```

**Naming base types** When the data source is a property path, each path element becomes a base type for the `FROM` data type. In this case, names for base type members correspond to property names in the path.

**Property path** Property path data sources define an inner join operation, i.e. the path in the example below provides all children that have one or more cars and its parent. Persons that do not got children or persons with children that do not have cars are not element of the `FROM` collection.

```
FROM ( Person.children.cars )
```

**Operation** Instead of a complex data type, one may define an operand as view source. When the data source is a result of an operation, a base type name has to be defined explicitly.

```
FROM ( set<Person> relatives = Person.GetRelatives )
```

One may omit the type definition, when the operation is defined in the resource data base or in a pre-loaded OSI function, which provides the type in the return value definition.

**Open views** When not defining `FROM` operands (omitting `FROM` clause), the view is considered to be an open view. Typically, open views are defined a local views for the data type they apply on. In case of open views, the view may be referenced with a preceding collection as data source by referring to the local view in the `SELECT` operation.

```
Persons.SELECT(PersView);
```

The view employment can apply to any `Person` collection, i.e. the `Persons` extent, but also on the `children` collection of 'Miller'.

When names have been assigned to source elements, view parameters can be referred to by name:

Data sources defined in the `FROM` clause define the scope for the properties and methods referenced in the `WHERE` expression as well as for the view member definitions and aggregation function sources for grouping definitions.

## Where

The `WHERE` clause describes a precondition for filtering data in the `FROM` data source. The filter condition is an expression returning true or false, which must be defined in terms of the `FROM` operation data type, i.e. the condition may refer to `FROM` data type properties, only. When a `WHERE` filter condition has been defined, instances in the `FROM` data source returning false will be ignored.

```
// select unemployed persons
FROM ( Person ) WHERE (company.count == 0)
```

## Group By

Grouping provides a way of aggregating data. The grouping clause defines the way, instances from the data source are grouped.

```
// Group Persons by age, sex
VIEW myView
FROM ( Persons )
GROUP BY (sex,
          string inc_group = (income < 1000 ? 'poor' :
                              income < 5000 ? 'medium':
                              income < 100000 ? 'rich':
                              'very rich' ),
          string age_group = (age < 20 ? 'young' :
                              age < 50 ? 'middle':
                              'old' ) )
{ ATTRIBUTE {
  string    age_group;
  string    inc_group;
  Sex       sex;
  INT(10,2) inc = sum(income);
  INT(10,2) avr_inc = average(income); }; };
```

A grouping is defined by an operand list, where each operand defines a grouping attribute (dimension). By default, dimensions also define the order key for grouping instances. `GROUP` operands should define attributes and are passed to the `GROUP` method.

Each `GROUP` instance contains grouping attributes (dimensions) and a collection `partition` containing all instances from the `FROM` data source matching the grouping attributes. The `partition` property becomes the default source for aggregation functions.

In case of `GROUP` definition in a view definition or `SELECT` statement, source operands for view members (`SELECT` parameters) refer to the `GROUP` data type. Aggregation functions without calling object refer to `partition`, i.e. property names passed to aggregation functions must be defined in the data type for partition instances, i.e. in the `FROM` data type.

#### Attributes

Operands in the operand list may be attribute names or property paths referring to attributes defined in the `FROM` data type (e.g. `(age, sex)`) or data type of calling object. Grouping attributes usually refer to elementary data types, but may also refer to complex or user-defined enumerations.

```
// Group by sex
GROUP BY ( age, sex )
```

#### Named attributes

By preceding the attribute references with names, one may explicitly assign attribute names to grouping attributes.

```
// Group by sex
GROUP BY ( age_group = age, sex )
```

#### Operation

When the data source is an operation, an attribute name has to be defined explicitly. One may omit the type definition, when the operation is defined in the resource data base or in a preloaded OSI function, which provides the type in the return value definition.

```
// Group by sex
GROUP BY (string age_group = GetAgeGroup(), sex )
```

Instead of referring to a function, one may also define inline expressions as data source.

#### Having

The `HAVING` clause describes a filter as post-condition for finally filtering data. The filter condition is an expression returning true or false, which must be defined in terms of the final view structure, i.e. to properties defined as view members in a view definition or `SELECT` clause.

When a `HAVING` filter condition has been defined, view instances returning false for the condition will be ignored.



```
// select persons by family income
VIEW myView
  FROM ( Persons )
  HAVING ( family_income > 100000 )
{ ATTRIBUTE {
  string name; string first_name; int age; Sex sex;
  int(10,2) family_income = GetFamilyInc(); }; };
```

## Aggregation functions

In case of grouping, view members may refer to default aggregation functions (see “Aggregation functions”) without preceding the function call with a collection name (calling object). In this case, `partition` is assumed as default calling object. Using default aggregation performs better than calling each aggregation function with the collection name, since the partition collection is read only once for evaluating all aggregation functions.

```
// Group Persons by age, sex
View myView
  FROM ( Persons )
  ORDER BY(string age_group = (age < 20 ? 'young' :
                               age < 60 ? 'middle':
                               'old' ) DESC,sex)
{ ATTRIBUTE { string age_group;
  int(10,2) tot_income = sum(income);
  int(10,2) avr_income = avarage(income);
  int(10,2) dev_income = deviation(income); }; };
```

## Order By

Ordering allows defining an instance order for created view instances. One may order instances by predefined key referring to a key name defined for the view data type or by order attributes.

```
// Order Persons by age_group
VIEW myView
  FROM ( Persons )
  ORDER BY(string age_group = (age < 20 ? 'young' :
                               age < 60 ? 'middle':
                               'old' ) DESC,sex)
{ ATTRIBUTE { string name; string first_name;
  int age; Sex sex; }; };
```

## Key order

For ordering the view by key, the order key has to be defined for the view data type (data type key definition). In this case, the `ORDER BY` expression simply contains the name of a defined key.

```
// order by keys 'pk'
ORDER BY ( pk )
```

## Attributes

Operands in the order operand list may be attribute names or property paths referring to attributes defined in the FROM data type (e.g. (age, sex) ) or data type of calling object. Grouping attributes usually refer to elementary data types, but may also refer to complex or user-defined enumerations.

```
// order by age, sex
ORDER BY ( age, sex )
```

## Named attributes

By preceding the attribute references with names, one may explicitly assign attribute names to grouping attributes.

```
// order by age sex
ORDER BY ( age_group = age, sex )
```

## Operation

When the data source for an order operand is an operation, an attribute name has to be defined explicitly. One may omit the type definition, when the operation is defined in the resource data base or in a preloaded OSI function, which provides the type in the return value definition.

```
// Order by age group, sex
ORDER BY (string age_group = GetAgeGroup(), sex )
```

## 5.6.2 Type Property List for Views

Type properties allow defining persistent views, i.e. for views that will store instances in the database. There is always a risk storing views in a database, since views are not updated automatically, when instances or collections the view is based on, do change.

Typically, external views are used for making a view semi-persistent, i.e. storing the view results in a transient or temporary extent, which is automatically deleted, when the application terminates.

The advantage of semi-persistent views is, that semi-persistent views support all features of persistent collections, as multiple sort orders, multiple access to view instances etc. That means, semi-persistent views can be accessed as ordinary extents or collections as long as the application is active.

### Type properties

Type or view properties are defined as key and extent definitions.

```
VIEW Employment
(
  KEY {
    IDENT_KEY ik_pid(pid, cid);
    sk_Name (cid, pid);
  };
  EXTENT ...
)
```

### Key definitions

Key definitions allow defining one or more keys, which can be used later on for creating indexes for collections to provide fast access.

When the view defines a `GROUP BY` clause, an identifying key with a temporary name `KEY_nnnnnnnnn` is automatically defined, which consists of one all attributes included in the in the `GROUP BY` clause.

Otherwise, one of the user-defined keys can be marked as identifying key, which usually determines the default order when accessing data in the view collection.

View key components may refer to defined view members, only.

### Extent definitions

For a view type any number of view extents can be defined. A view extent is automatically filled (evaluated) when being accessed or opened..

```
VIEW Employment
(
  ...
  EXTENT TRANSIENT Employments;
)
```

Usually, view extents are transient, i.e. they are built, when referring to (opening) a view extent. One may, however, store extents to the database. This might be useful, when an extent is used many times and when the content of the extent (i.e. the source collections) do not change.

One may, however define view extents globally by referring to the view definition as data type for the extent. In both cases, data source(s) for the view must have been defined as global access paths

In order to use local views defined in the context of a complex data type, one may define transient collection properties for the data type that refer to the local view definition as data type. In this case, the view collection will be updates for the currently selected instance when being accessed.

```
VIEW Employment
(
  ...
  EXTENT Employments ORDERED_BY( ik_pid UNIQUE, sk_name );
)
```

When defining persistent view extents, at least one unique sort key must be defined.

### 5.6.3 View Members

View members (*view\_member*) are the properties, the view consists of. View properties are either attributes or references. You cannot define relationships or base structures (inheritance) for a view definition. In order to refer to another view, one may refer to an appropriate view collection (extent or property) in the `FROM` clause.

Beside view properties (attributes and references), the view allows defining methods, that operate on the complex data type defined by the view.

When the view definition contains a `GROUP BY` clause, a `partition` collection property (transient reference) is implicitly defined as view property.

**View source** The source instance for evaluating view properties is either the `GROUP BY` instance (when `GROUP BY` clause has been defined) or the `FROM` instance for views without grouping.

**Transient view members** View members defined as transient members are not considered as view members evaluated on base of view source, but allow defining post calculations, i.e. those are evaluated in the context of the view instance, i.e. evaluation functions or expressions may refer to view members, only.

**Attributes** Attributes are defined similar to attributes in the complex data type definitions (class or interface). Attributes usually require a source operand which may be passed as `SOURCE` or initial value based on the complex data type of the source defined in the `FROM` or in `GROUP BY` clause.

```
VIEW Retired FROM ( Person )
    FROM ( Person ) WHERE (company.count == 0)
    HAVING (age_years) >= 65 )
{
    ATTRIBUTE {
        INT(10,2) ch_income = children.sum(income);
        INT      ch_count  = children.count();
        INT(3)   age_years SOURCE( (Date - birth_date).year );
        TRANSIENT INT(10,2) ch_avr_inc = income / ch_count;
        ...     };
};
```

When not defining an attribute source, the attribute name is considered to be an operand, i.e. it refers to a member with the same name in the view source data type. The shortest way for defining a view member is defining just an attribute name preceded by a type definition, which is also considered to be an operand based on the source data type.

```
VIEW Adult FROM (PERSON) WHERE ( age > 18 )
{
  ATTRIBUTE {
    STRING(40) name;
    STRING(40) first_name;
  };
};
```

When no data source is defined, the member name is assumed to be the operand name, as well, i.e. the view member refers to the member in the source data type with the same name.

View attribute definitions follow the general rules for defining attributes and references (see chapter “Attributes”). Specific details for the assignment operand are described in “Query operations – SELECT”.

## References

Reference collections are defined similar to references for other complex data type definitions (class or interface). The difference is, that each reference requires an initial value or SOURCE operand, which is an expression or function based on the complex data type of the view source. The result of the expression must be a complex data type.

```
VIEW Grown_Up_Children FROM ( Person )
{
  ...
  REFERENCE SET<Person> adults = children.WHERE(age > 18);
};
```

One may also refer to view definitions as complex data type for a reference.

```
VIEW Adult FROM (Person) WHERE ( age > 18 )
{
  ATTRIBUTE {
    STRING(40) name;
    STRING(40) first_name;
  };
  REFERENCE SET<Person> young_parents = parents.YoungParents();
};
```

## 5.7 Aggregation schema

An aggregation schema is an extended view definition. In contrast to a view definition, the collection resulting from an applied view definition contains instances on different hierarchy levels, while an ordinary view provides (aggregated) instances on the lowest aggregationlevel defined by the `GROUP` clause.

A view definition becomes an aggregation schema by adding an aggregation type to the view definition:

```
VIEW vname [from_spec] [where_spec] [group_spec] [having_spec]
          aggr_option_list [type_property_list]
{
  [ view_member(*) ]
};
```

The position for the aggregation option may be at any place in the view header.

```
VIEW M<Aggr ( INT(10,2) time = sum(duration) )
FROM (TimeSheet) AGGREGATE(simple)
GROUP BY ( STRING(10) personId = person(0).id,
          STRING(10) projectId = task(0).GetProject().id,
          IDate      iDate = GetIDate(start) );
```

Similar as defining an aggregation schema within the database dictionary, one may use an ad-hoc aggregation schema using the `SELECT` statement. The example in the following sub-chapter refers to the `SELECT` statement, that is schema definition and execution at once.

### 5.7.1 Aggregation example

The following chapter describes the aggregation example as being provided with the PMA example application.

The PMA example is based on a project management system that stores time sheets per project and person. The grouping level is `projectId (STRING)`, `personId (STRING)`, `iDate (IDate)`. `IDate` is a complex grouping dimension derived from date, which has got three attributes (`year`, `month` and `day`). Complex dimension attributes provide aggregation levels for each subordinated dimension attribute, i.e. aggregation levels provided are `year`, `year, month` and `year, month, day`.

The aggregated variable is the `time` spent on a certain project or by a certain person or in a certain time interval (`time sheet duration`).

```
STRUCT IDate {
    INT(2) year;
    INT(2) month;
    INT(2) day; };

SELECT ( INT(10,2) time = sum(duration) ) HIERARCHY(Delayed)
FROM (TimeSheet)
GROUP BY ( STRING(10) personId = person(0).id,
          STRING(10) projectId = task(0).GetProject().id,
          IDate iDate = GetIDate(start) );
```

**Notes:** `GetProject()` provides the project a task in a hierarchical task structure belongs to. This may also be the project itself, which also is a task.

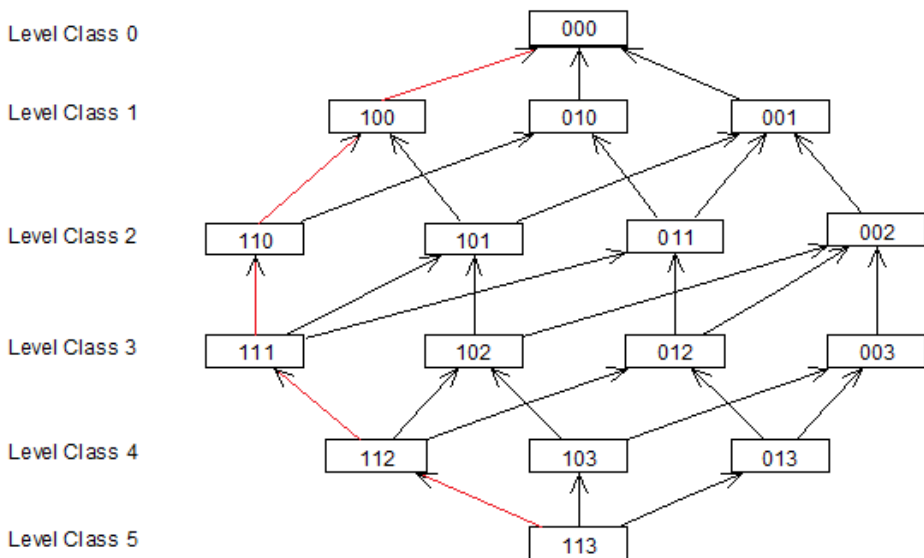
`GetIDate()` is a function that creates an `IDate` structure from a database date value.



## 5.7.2 Aggregation levels

Aggregation levels defined by means of an aggregation schema contain view instances aggregated on different levels. A higher aggregation level is always created by reducing one of the dimensions in the dimension level.

`personID` and `projectID` are elementary dimensions, that only provide dimension level 1 and 0 (dimension has no value on this level). `iDate` provides dimension levels 3 (year, month, day), 2 (year, month), 1 (year) and 0. Thus, the possible aggregation levels in this example look like following:



Eg. Level 112 means level dimensions `personID`, `projectID`, (year, month). The numbers identifying an aggregation level are called **level identifier**. In an aggregation collection, level identifier become an attribute of aggregated instances. Maximum 8 aggregation dimensions with maximum 9 dimension levels for each dimension are supported.

The red line in the picture shows the aggregation levels for a **simple** aggregation. Arrows between level dimensions point to possible next higher dimensions in the aggregation schema. In case of **delayed** aggregation, only the view level (113 in the example) will be created. Other level are created when being requested, i.e. when being accessed.

When accessing a level that is not the next higher level of an aggregation level already created, the system looks for an optimal solution.

In case of **complete** aggregation, all aggregation levels are created when creating the aggregation collection.

### 5.7.3 Accessing aggregation collections

Because of different aggregation levels and specific relationships stored in an aggregation collection, aggregation collections become much more complex than ordinary collections.

In order to provide access to a certain level within an aggregation collection and to subordinated instances, additional functionality is provided for an aggregation collection by the **View** access class. The class inherits from **Property** but provides additional functionality for views. Those functions are available in general for each kind of view result, but for ordinary views they do not make much sense.

#### **View functions**

Extended view functionality provides several service functions with a default implementation that might be overloaded in an aggregation schema. Extended functionality supports special behavior resulting from the level identifier:

- `view` - Get view handle from **Property** instance
- `levelCollection` - Get all instances for a certain level identifier
- `levelPartition` - Get level partition relationship (subset for an aggregation instance on next lower level - *drill down*)
- `levelParent` - Get levelParent relationship (parent instances for a passed aggregation level)
- `levelNumber` - Get level number from level identifier (character level identifier converted to integer)
- `levelValue` - Provide level value (dimension attributes)
- `levelName` - Get hierarchy level name for selected aggregation instance
- `levelIdentifierName` - Get hierarchy level name for passed level identifier
- `levelIdentifier` - Get level identifier
- `levelDisplayValue` - Get level display value for level attributes

## User functions

In order to support user names for levels and keys, the application may implement service functions in the view class that overload functions named above:

- `LevelName` - Provide hierarchy level name for selected aggregation instance
- `LevelIdentifierName` - Provide hierarchy level name for passed level identifier
- `LevelDisplayValue` - Provide level display value

In addition, the view class may implement an application specific `Aggregate` function, which provides user defined aggregation algorithms for all view members, that do not have got a source definition and which are not transient.

## 5.8 Template Data Types

Template data types are kind of meta-types describing the way one or more instances of a given type are managed. Within OSI, template data types are typically collection data types, but other template data types can be defined as well.

Template data types are defined in general as:

```
template_type_name < simple_type_spec >
```

OSI allows the definition of any template data type (e.g. when defining C++ members) for being loaded into the dictionary, but OSI supports only a limited set of template types according to the ODMG recommendations when running OSI functions.

### Collection types

Four types of collections are recommended by ODMG:

- Set
- List
- Bag
- Array

These four types differ in order and unique constraint. Both, order and unique constraints are handled as index options in ODABA and not as collection options. Thus, OSI supports the four template types for collections, but it will not differ between them.

```
SET<Person> children; // same as BAG<Person> or LIST<Person>
```

Collection types are typically used for defining references and relationships. But collection types can be referred to also in member definitions for structure members, attributes, parameters or variables. In this case, collection types define transient data and can be used in different ways.

One way is defining references to collections stored somewhere else. The other possibility is creating transient collections.

**Standard template types**

Besides collection types OSI supports string template types, which are handled the same way as OSI `STRING` type. Other standard template types as `INTERVALL`, `VECTOR` etc. can be defined, but are not supported in OSI functions. OSI supports those definitions in order to support other program languages as C++ or Java when loading schema definitions to the dictionary.

## 5.9 Member Definitions

Member definitions have been referenced when defining structures, interfaces, classes and views. Practically, the definition of members is always the same, i.e. one may define base collections for a relationship in an interface definition, even though this will not have any effect.

In the previous chapters, the features supported for members defined in different contexts, have been explained. Here, the maximum features of member definitions will be discussed.

Members are the elements, a structure, interface, class or view consists of. There are different categories of members, that can be defined:

- (Inheritance/extends)
- Constants
- Types
- Exceptions
- Members
- Attributes
- References
- Relationships
- (Keys)
- Methods
  - (C++ Functions)
  - OSI Functions
  - (Templates)
  - (Forms)

### OSI specific

The current version of OSI/ODL accepts constants, exception and type declarations, but does not support those in OSI utilities. When such declarations are defined in an ODL or OSI file, they will be ignored.

When running an OSI script file, OSI function is the only type of methods, which is supported. When loading an ODL schema, methods are considered by default as C++ functions. Other method types must be declared explicitly.

Inheritance/extents specifications are, according to [ODMG], not explicitly defined as members. In ODABA, however, inheritance or extents specifications are considered as members (specialized relationships) and can be accessed as such. For compatibility reasons, they are defined, however, in the corresponding headers (class, interface, view).

Support for templates and forms is planned for OSI version 2.0. In OSI 1.0 you may refer to forms only, which are stored in the resource database (dictionary).

Member type keywords (attribute, reference, relationship) can be placed in front of each member definition or once for a block of interface members.



## 5.9.1 Inheritance

Inheritance describes the intensional specialization of a complex data type. Since OSI always supports both, intensional and extensional member specifications, the inheritance specification supports also extensional features.

```
... [ inheritance_spec | extends_spec ]
```

OSI inheritance specifications are supported as `EXTENTS` specification for classes, but also as the inheritance specification, i.e. you may use the inheritance operator `⋮` as well as the `EXTENDS` keyword followed by one or more base type references.

```
CLASS Employee : PUBLIC Person person BASED_ON Persons  
                    INVERSE pers_ref
```

When inheriting from more than one base type, base type references are separated by comma.

Since OSI considers inheritance specifications as member definition (specialized relationships), inheritance specifications may get a name (person in the example above), which allows distinguishing between different bases types of the same type. Moreover, several additional attributes are supported for inheritance specifications, which are inherited from relationship definitions.

## 5.9.2 Members

Member definitions describe properties of a complex data type. Member definitions are used in structure definitions, but also for defining variables or parameters. The general definition for a member is:

```
domain_type declarators;
```

In variable declarations or within a structure definition, a member definition may contain one or more declarators. As parameter, only one declarator per member definition is allowed.

### Data type

The domain type or data type defines a basic (literal) type or a user defined type (enumeration or complex data type as class, interface or structure). Data types supported by OSI are described in detail in chapter "Data Types". The following examples illustrate the use of data types in member definitions.

### Text types

Text type members are `STRING`, `WSTRING`, `CHAR` and `CCHAR` (coded character). The difference between `CHAR` and `STRING` types is that `CHAR` types are filled with spaces, while `STRING` types are 0-terminated.

Text type members usually have a size limitation, which defines the maximum size for the value (default: 256).

```
CHAR(16) pid;  
STRING(40) name; // same as string<40> name;
```

There is a difference between size and array definitions. The size is always defined in connection with the type (as in the example above)

Array dimensions are defined in connection with the declarator.

```
CHAR(16) pid[5];  
STRING name[3]; // array wir 3 name dynamic strings;
```

When defining size and dimension, an array of strings or character values with the given size is defined.

Text type members always reserve the defined number of bytes in the stored instance when being defined as members or variable. Thus, large text fields should be stored as `STRING` reference or `MEMO` member.

String definitions without size specification are allocated as dynamic string variables, which allocate size on requirement.

**Numeric types** In principal, OSI differs between floating (`REAL`) and integer (`INT`, `UINT`) types, i.e. each numerical value is stored either as float or as integer.

Numeric types do have a size, which defines the maximum number of significant figures stored in the number (e.g. size 5 for 100.01)

```
REAL    income; // same as double income;
INT(3)  age;     // same as short age;
```

Depending on the size, the type corresponds to the appropriate standard type (float, double, short, long, long long), which can also be used in the definition.

OSI supports decimal types (`DEC`), which are considered as integer with a precision factor. `DEC` and `INT` definitions are equivalent.

```
DEC(7,2) income; // same as INT(7,2) income;
```

The factor can be negative, in which case the stored value is multiplied with an appropriate factor (e.g. with 1000 for `DEC(5, -3)`).

**Date/time types** Date (`DATE`) and time (`TIME`) can be defined as well as time stamps (`DATETIME`). `DATE` and `TIME` are considered as integer type and can operate with integer values. A size specification for `DATE` or `TIME` influences the default presentation of the member, but not the way the value is stored.

```
DATE(6)  birth_date; // 53/06/30
DATE(8)  birth_date; // 1953/06/30
```

Timestamp (`DATETIME`) is a structured field containing a date and a time value.

```
DATETIME last_update;
...
last_update.date = SystemClass::Date();
last_update.time = SystemClass::Time();
```

Date and a time value can be accessed as structure members of the `DATETIME` structure.

**Other literal types** Other literal types supported are `BIT`, `BOOL`, and `ANY` (or `VOID`). `VOID` is valid for references and relationships and pointer members, only.

**Complex data types** Members may also refer to complex (user-defined) data types, which might be a structure or class type.

```
Address location;
...
Print("City :" + location.city);
```

In this case, members of the related complex data type can be accessed via appropriate property paths.

### Template data types

Members may also refer to template data types, especially collection data types. Other template data types can be defined, but cannot be processed in an OSI function, except string template types, which are handled the same way as `STRING` data type.

```
SET<Person> grand_children;
```

Referring to a collection type in a member definition defines always a reference to a collection, but not the collection itself. For defining a collection, you may define an extent (global collection) or a reference or relationship.

Assigning a value (collection) to a collection member defined by value will create a separate cursor for accessing the collection.

```
SET<Person> my_persons = your_persons;
```

In the example above, instances selected in `my_persons` collection and `your_persons` collection may differ.

```
SET<Person> &my_persons = your_persons;
```

When defining collection type members by reference, `my_persons` will share the cursor with `your_persons`, i.e. changing the selection in one collection variable automatically changes the selection in the other variable.

### Declarators

Declarators will list one or more member names with their qualifiers.

```
[ref_symbol(*)] identifier [fixed_array_size(*)]
                    [assigned_value]
```

### Reference type

Declarators may be preceded by a reference type. In OSI, the '`&`' symbol is allowed, only, to indicate by reference properties (transient attributes, variables, structure members, parameters). Loading an ODL schema, pointers of any level can be defined as transient fields. Thus, '`*`' or '`**`' are valid reference types as well for defining C++ methods, but those are not supported in OSI functions.

"by reference" members must be defined as transient when being used as class attributes.

## Arrays

By default, member declarators have dimension 1. By defining a fixed array size, array members can be defined.

```
Address location[3]; // reserves three addresses
```

Array members are considered as collections and can be accessed by index.

```
address = location(3); // same as: address=location.get(3);  
address = location[3]; // same as: address=location.provide(3);
```

OSI supports two access operators for collections, which operate different for collection types. For arrays, both operands works in the same way providing the instance at the defined position.

For text types, the dimension specification is interpreted as size, as long as no size has been defined.

Array members will reserve space for all array elements in the object instance, i.e. OSI does not support dynamic array members. For defining dynamic arrays, a reference or a collection data type should be defined rather than a array member.

## Initializing members

For initializing members, an initial value or an expression can be assigned to the member. The time point for initializing a member depends on the context, in which the member is defined (as variable, structure member, parameter or attribute).

```
Sex sex = 'female';
```

Instead of assigning constants or expressions, one may assign OIF data strings for complex variables. When the data type is a C++ class data type providing an OSI interface, one may also initialize variables by calling the constructor of the class. A constructor is a class member function with the class name as function name.

```
Color my_color(100,100,100);
```

Constructors might be provided also for persistent classes by implementing an OSI function with the same name as the data type. One may also call constructors from within the function code.

```
VARIABLES  
Color my_color; // create variable  
PROCESS  
my_color.Color(100,100,100); // initialize variable
```

### 5.9.3 Attributes

Attribute definitions describe properties of a complex data type, which always exist in an object instance. The general definition for an attribute is:

```
[type_ref_options(*)] domain_type declarators;
```

Structure types consist only of attributes, while other complex data types may define different member types. In this case, attributes must be marked as such in the definition.

```
ATTRIBUTE {  
    STRING(40)    name;  
    STRING(40)    first_name;  
};  
ATTRIBUTE INT(7,2)    income;
```

You may mark each attribute separately or define an attribute block, but you may also mix both ways..

#### Type options

The type reference defines the type (or generic type) and additional properties (as size and precision), when available for the type.

Several options can be defined for an attribute. Options are defined before the declarator list and apply to all declarators in the list. .

#### TRANSIENT

Transient attributes can be defined as direct (imbedded) members or as "by reference" members. Attributes are transient by default, when being defined in a structure or interface or in a transient class, i.e. in a class that does not create persistent instances. In persistent classes, transient attributes are available within the application, but are not stored with the instance to the database. For persistent classes, transient attributes must be defined explicitly.

```
ATTRIBUTE TRANSIENT INT(3) age = (Date - birth_date).year;
```

#### STATIC

Static attributes can be defined for transient instances or as transient attribute in persistent instances, i.e. ODABA considers static fields by default as transient.

Thus, one could add a static attribute to the Person class:

```
ATTRIBUTE STATIC TRANSIENT INT no_of_females;
```

Static attributes are created only once for all class instances.

**NOT\_EMPTY** The `NOT_EMPTY` option defines a default constraint. Defining an attribute as `NOT_EMPTY` will deny storing instances, for which the attribute value is empty. The empty state is defined depending on the data type.

```
ATTRIBUTE NOT_EMPTY STRING(40) name;
```

**VIRTUAL** Virtual properties are properties, which can be redefined in specialized data types. Redefinition includes changing the type or the size of the property, but not the property type, i.e. one cannot redefine an attribute as a relationship.

**privilege option** Privilege options define the access rights to the property. So far, this feature is not supported by ODABA, i.e. ODABA considers all properties as public. This will be changed, however, in version ODABA 10. Hence, it is suggested to set the option properly.

Supported privilege options are:

- PRIVATE
- PROTECTED
- PUBLIC

In contrast to C++, this qualifier must be defined for each property separately.

Default: PROTECTED

```
ATTRIBUTE PRIVATE DATE birth_date;
```

**Data type** The domain type or data type defines a basic (literal) type or a user defined type (enumeration or complex data type as class, interface or structure).

Attributes may refer to all data types supported for members. When referring to collection data types, the attribute must be marked as transient or it must be defined in a transient reference.

**Declarators** Declarators will list one or more attribute names with their qualifiers. Attribute declarators are member declarators with a constraint extension.

```
[ref_symbol(*)] identifier [fixed_array_size(*)]  
[assigned_value] [constraint]
```

Details for member declarators are described in chapter "Members".

## Initializing attributes

For initializing attributes an initial value or an expression can be assigned to the attribute. The assigned value is computed and set, when a new object instance is created for the complex data type, which defines the attribute.

```
ATTRIBUTE Sex sex = 'female';
```

When defining initial values for transient attributes in persistent instances, the initial value is computed and set always, when an instance is read. For transient data types the member is initialized, when the object instance is constructed,

```
ATTRIBUTE INT(3) age = (Date - birth_date).year;
```

When defining an expression as assigned value, this must be a valid expression (operand) within the class definition, i.e. the expression may refer to global variables as well as to class variables.

## Constraint

Constraints allow defining rules for validity checks on the attribute.

```
ATTRIBUTE DATE birth_date  
CONSTRAINT(birth_date > '1879/12/31');
```

The constraint expression (operand) must be a valid expression within the class definition, i.e. the expression may refer to global variables as well as to class variables.



## 5.9.4 References

References describe details belonging to the object. References can be defined in class definitions, only. References may refer to complex data types, text types (`STRING`) and binary large objects (`BLOB`). References cannot refer to any literal data type except `STRING` and `BLOB`.

The general definition for a reference is:

```
[type_ref_options(*)] domain_type  
    [ref_option(*)] ref_declarator(s)
```

A reference may refer to one or more instances of a given type. A singular reference just refers to the type:

```
REFERENCE Address location;
```

Multiple references are defined as template data types or by defining a dimension. The following specifications have the same effect:

```
REFERENCE SET<Address> locations;  
REFERENCE Address locations[];  
REFERENCE Address locations[0];
```

Using template types corresponds to the ODMG suggestions, while the specifying a dimension is a specific ODABA feature, which allows limiting the number of instances in a collection.

```
REFERENCE Address locations[3];
```

### Type options

The options (`type_ref_options`), which can be defined for a reference, are similar as for attributes, except `static`, which is not supported.

`TRANSIENT`

A transient reference in a persistent structure allows computing collections for each instance at run-time. Thus, you can avoid storing redundant information, which can be calculated at runtime.

`VIRTUAL`

Virtual references can be overloaded in a specialization of a class definition. Thus, specializing the type allows specializing the type of referenced objects. Type specialization for virtual references must inherit from the reference data type in the base class.

`NOT_EMPTY` The `NOT_EMPTY` constraint for a collection accepts empty collections until the first instance is added to the collection. Later on, the collection can never become empty. Defining `NOT_EMPTY` for a singular reference allows associating an instance only once, i.e. one should never use this option for singular references.

privilege option Privilege options define the access rights to the reference. So far, this feature is not supported by ODABA, i.e. ODABA considers all properties as public. This will be changed, however, in version ODABA 10. Hence, it is suggested to set the option properly.

Supported privilege options are:

- `PRIVATE`
- `PROTECTED`
- `PUBLIC`

In contrast to C++, this qualifier must be defined for each property separately.

Default: `PROTECTED`

**Collection options** Collection options (`ref_option`) can be set to enable specific features for an ODABA collection. Collection options make sense for collections defined in classes. Collection options are defined before the declarator list and apply to all declarators in the list.

`GUID` The `GUID` option indicates that global unique identifiers (GUID) are to be created for each instance, which is created for the reference collection. This option need to be set only, when the referenced data type did not define the `GUID` option.

GUIDs can be created for classes, only, which inherit from `OBJECT`. Otherwise, the option is ignored.

```
REFERENCE Address GUID location;
```

`WEAK_TYPED` A reference may refer to instances of different types, which are typically specializations of a common base type. For weak-typed references, the data type for the reference defines the common base type for the instances in the collection.

```
REFERENCE Address WEAK_TYPED location;
```

In this case, the reference can store `CompanyAddress` or `PersonAddress` in the collection, which are supposed to be specializations for `Address`.

For creating an instance of a given type, the type must be set for the collection in advance.

```
REFERENCE BAG<Address> WEAK_TYPED location;  
...  
location.setType('CompanyAddress');  
location.insert; // creates a company address
```

**DELETE\_EMPTY** For avoiding the deletion of instances, which refer to other objects in the collection, the `DELETE_EMPTY` option can be set. In this case, an instance of the class can be deleted only, when the referenced collection is empty.

```
REFERENCE Address DELETE_EMPTY location[3];
```

**MULTIPLE\_KEY** When changing from single ordered collection to a multiple ordered collection, a new schema version must be created for the database. This can be avoided when setting the `MULTIPLE_KEY` option for single ordered collections.

```
REFERENCE Address MULTIPLE_KEY location[3];
```

**UPDATE** The `UPDATE` option indicates, that instances can be inserted to or deleted from the reference, without updating the instance containing the reference. This option increases the accessibility in multi-user environments, since several users may update the collection without getting conflicts.

```
REFERENCE Address UPDATE location[3];
```

**Data types** The domain type or data type defines a basic (literal) type or a complex data type (class type).

**Text types** References can be used to refer to large text fields (`STRING`, `MEMO`). When storing large text fields as references, the space reserved for the text field depends on its current length.

```
REFERENCE STRING(4000) notes;
```

The size passed to the type defines the maximum number of characters in the field. Array dimension are not supported for text references, i.e. text references are always singular.

Binary large objects      References can store binary large objects (BLOB), e.g. images or movies. BLOB properties can only be defined as references.

```
REFERENCE BLOB image;
```

No maximum size can be defined for a BLOB reference. Array dimension are not supported for BLOB references, i.e. BLOB references are always singular.

Any type      You may define an untyped reference, which may refer to instances of any data type (VOID or ANY). Defining an untyped reference, the first instance added to the collection determines the type of the reference, i.e. an untyped reference may contain instances of any type, but all instances are of the same type.

```
REFERENCE VOID objects[]; // same as: any objects;
```

For defining a reference containing a mixture of instances of any type, the reference must be defined as weak-typed reference.

```
REFERENCE VOID WEAK_TYPED objects[];
```

Complex data types      Typically, references refer to object instances of complex data types (user-defined types). References to enumeration types are not allowed.

```
REFERENCE Address location;
```

By default, references are singular, i.e. it may refer to maximum one object instance.

Template data types      Typically, references are defined as collection data types. Since ODABA does not distinguish between array, list, set and bag, collection references are usually defined a dimension value, but any of the collection types could be used as well.

**Declarators**      The reference type definition is followed by one or more reference declarators. Reference declarators allow defining one or more reference names (identifier) appended by several qualifiers.

```
identifier [col_dimension] [assigned_value] [constraint]
[order_keys]
```

Dimension      Multiple references can be defined (for compatibility reasons) in different ways. OSI supports the explicit definition of a maximum number of object instances in a reference collection.

Default: 1

```
REFERENCE Address location[3];
```

For allowing an unlimited number of referenced object instances, you may define 0 or an empty dimension value as dimension or use one of the standard collection types.

```
REFERENCE Address location[0];  
REFERENCE Address location[];  
REFERENCE SET<Address> location;  
REFERENCE LIST<Address> location;  
REFERENCE BAG<Address> location;
```

The difference between **SET**, **LIST** and **BAG** is the sort order and unique property. Since ODABA provides separate features for defining one or more sort orders for a collection in an index definition, the four examples above are equivalent. More detailed definition for different sort orders, unique instances or keys etc. can be better defined in a number of collection options. Nevertheless, you may use the standard collection types **SET**, **LIST** and **BAG** for defining your collections. Using **LIST**, **SET** or **BAG**, you cannot define a dimension value.

#### Initial value

The same way as for attributes, you may define an initial value for a reference. This makes sense, however, only for transient references, which can be initialized this way when reading or creating a new instance.

The initial value for a reference is an expression (operand), which must be valid in the context of the class, i.e. it may refer to class members and global variables.

#### Constraint

Constraints allow defining rules for validity checks on the reference. The constraint is a collection constraint, i.e. it operates on the whole collection and not on a single instance. Thus, constraints cannot be used for controlling deletion or insertion of instances. This is handled by reacting on corresponding system events.

The constraint for a reference is an expression (operand), which must be valid in the context of the class, i.e. it may refer to class members and global variables.

#### Order keys

A reference allows defining one or more order keys. How to define keys and order keys is described in "Keys and key references".

## 5.9.5 Relationships

Relationships describe links to related object instances. Relationships are special references and inherit all options and attributes described for the reference. Relationships can be defined in interfaces (limited specification) and in classes.

The general definition for an relationship is:

```
[type_ref_options(*)] domain_type  
    [rel_option(*)] rel_declarator(s)
```

Relationship can define single or multiple references to instances of complex data type. Relationships cannot refer to literal data types (base types or enumerated types).

### Type options

The options (type\_ref\_options), which can be defined for a relationship, are the same as for references (see “Relationships – Type options”)

### Collection options

In addition to the collection options defined for references (see “References – Collection options”), several additional collection options can be defined for relationships. These collection options make sense only for relationships in persistent data types (class definitions).

#### OWNER

A relationship can be defined as the owner of the instances referring to. When removing an instance from a owning collection, the instance will be automatically deleted.

Not owning relationships should always have a superset (BASED\_ON qualifier).

In an ODABA database, each instance belongs to exactly one owning collection.

```
RELATIONSHIP SET<Car> OWNER cars ORDERED_BY (ik_cid UNIQUE)  
    INVERSE company;
```

Here, the relationship cars (in Company) is the owner of the car instances in the collection, i.e. removing a car from the collection will delete the car instance.

#### NO\_CREATE

This option prevents instances from being created in a collection. When the option is set, only existing instances can be added to the collection. The option usually requires a superset (BASED\_ON qualifier)

This option should never be set for owning relationships, because one cannot create instances at all in this case.

```

RELATIONSHIP Car NO_CREATE used_cars[2]
                    BASED_ON company.cars
                    ORDERED_BY (ik_cid UNIQUE)
                    INVERSE users;

```

A person can use up to two cars from the company cars collection. When not defining `NO_CREATE`, associating a car with the person that does not exist yet in the companies car set, a new car would automatically created and added to the companies car set as well as to the used cars for the person. When defining `NO_CREATE`, no car instance is created in this case and associating a car to the person will be denied.

#### DEPENDENT

Instances may depend on a relationship without being owned by the collection. Being dependent causes deletion of instances removed from the collection (relationship). Thus, dependent has a similar effect as the `OWNER` option. `DEPENDENT` need not to be defined, when the relationship owns the instances (`OWNER`).

```

RELATIONSHIP SET<Car> DEPENDENT cars
                    BASED_ON Cars
                    ORDERED_BY (ik_cid UNIQUE)
                    INVERSE company;

```

Instead of defining cars as owned by the company, one may define the company cars as based on the Cars extent to create a collection containing all cars. In this case, the `DEPENDENT` option can be used to delete car instances, which are removed from a company's cars collection.

#### SECONDARY

When defining relationships with an inverse reference, deep copy functions may end up in an infinite recursion loop. The `SECONDARY` option indicates that instances for the relationship are not to be copied.

When defining a relationship pair, exactly one of the relationships should be marked as `SECONDARY`.

```

RELATIONSHIP Person children[0]
                    BASED_ON Persons
                    ORDERED_BY (sk_name UNIQUE)
                    INVERSE parents;
RELATIONSHIP Person SECONDARY parents[2]
                    BASED_ON Persons
                    ORDERED_BY (sk_name UNIQUE)
                    INVERSE children;

```

In this example, children are copied and parents are maintained automatically.

SHARED

Shared collections are collections, which can be shared between different transactions. This requires specific key locking strategies to avoid assigning duplicate keys to unique indexes. Since this causes loss of performance, this option should be set, when necessary, only.

```
RELATIONSHIP CAR SHARED location[3];
```

**Data types**

As domain type or data type for relationships, only class or interface types are allowed. Structure types, view types or enumerated data types cannot be used in as type for instances in a relationship.

**Declarators**

The relation type definition is followed by one or more relation declarators (*rel\_declarators*). Relation declarators allow defining one or more relation names (identifier) appended by several qualifiers. All reference qualifiers can apply on relationship definitions as well. Reference qualifiers are described in the Reference topic, except order keys, which are described below.

```
identifier [col_dimension] [constraint]  
[base_collection] [inverse_spec] [order_keys]
```

**Constraint**

Constraints allow defining rules for validity checks on the relationship. The constraint is a collection constraint, i.e. it operates on the whole collection and not on a single instance. Thus, constraints cannot be used for controlling deletion or insertion of instances. This is handles by reacting on corresponding system events.

The constraint for a reference is an expression (operand), which must be valid in the context of the class, i.e. it may refer to class members and global variables.



base collection

`BASED_ON` qualifier allows defining a superset (base collection) for the collection. For singular relationships, the superset defines a value domain for the instances referenced. For multiple relationships, the superset contains all the instances, which can be associated with the relationship.

Usually, each primary relationship (not marked as `SECONDARY`), which does not own its instances (`OWNER` option not defined), should have a base collection definition to define, who is the owner of the instances in the relationship.

When defining a base collection, an order key is required that is also order key in the referenced base collection (superset).

Typically, extents are defined as base collections.

```
RELATIONSHIP Person children[0]
                BASED_ON Persons
                ORDERED_BY (sk_name UNIQUE)
                INVERSE parents;
```

Here, children of a Person belong to the Persons extent, i.e. each person referenced in the children relationship is member of the Person extent as well, which owns the Person instances.

In some cases, it makes sense defining a related local collection (relationship) as superset.

```
RELATIONSHIP Car NO_CREATE used_cars[2]
                BASED_ON company.cars
                ORDERED_BY (ik_cid UNIQUE)
                INVERSE users;
```

In this case, the cars used by a Person belong to the cars collection of the company, the person works for. This means, different persons may have different base collections depending on the company associated with the person. When the person changes the company, it automatically loses all assigned cars, which belong to the previous company.

Defining a relative base collection as access path consisting of a number of relationships (traversal path), the following conditions must be fulfilled:

- a) each relationship in the path must have an inverse reference
- b) Each relationship in the path except the last one must be singular

For weak-typed relationships, one may define "\*" as base collection (BASED\_ON \*). In this case, the base collection depends on the type set for the relationship, i.e. the default extent for the selected type is used as base collection. The default extent is the extent with the same name as the referenced class (data type).

To maintain the superset relation properly, ODABA automatically adds instances to the base collection, which are created for the relationship and do not exist in the base collection. This can be avoided by setting the NO\_CREATE option, in which case creation of new instances is denied for the relationship.

**Inverse Relationship**

The INVERSE qualifier defines the inverse relationship for the current relationship. The inverse relationship must be a relationship property defined in the referenced type or one of its base types.

Relationships (except owning relationships) should define an inverse relationship (this is not a question of database consistency, but a question of good design). Relationships without inverse reference will cause problems when being referenced in property paths for relative base collections.

```

RELATIONSHIP Person children[0]
                BASED_ON Persons
                ORDERED_BY (sk_name UNIQUE)
                INVERSE parents;
    
```

According to ODMG suggestions [1], one may use scoped names for the inverse definition, but this is redundant and not required in OSI/ODL.

```

RELATIONSHIP Person children[0]
                BASED_ON Persons
                ORDERED_BY (sk_name UNIQUE)
                INVERSE Person::parents;
    
```

**Order keys**

A relationship allows defining one or more order keys. How to define keys and order keys is described in "Keys and key references".

```
RELATIONSHIP Car NO_CREATE used_cars[2]
                BASED_ON company.cars
                ORDERED_BY (ik_cid UNIQUE)
                INVERSE users;
```

In this case, the cars used by a Person belong to the cars collection of the company, the person works for. This means, different persons may have different base collections depending on the company associated with the person. When the person changes the company, it automatically loses all assigned cars, which belong to the previous company.

## 5.9.6 Keys and Key References

Keys are considered as structure or class members in OSI/ODABA. A key defines a structure consisting of one or more attributes, which are defined in the complex data type (class or view) the key belongs to.

Each class or view definition may contain any number of key members. Keys are the pre-requisite for defining collection indexes for extents, references or relationships in the database, but also for maintaining set relations.

```
CLASS Person
(
  KEY {
    IDENT_KEY ik_pid(pid);
    sk_Name (name,first_name);
  };
  ...
)
```

### Key definitions

Key definitions are preceded by the `KEY` keyword. You may use the `KEY` keyword for each key definition or define a number of keys in a `KEY` block.

### IDENT\_KEY

One of the keys can be marked as identifying key, which usually determines the default order when accessing data in a collection. The identifying key is marked by the `IDENT_KEY` option.

Defining extents for a class or view requires an identifying key, always. Identifying keys are also used for maintaining set relations.

### Key components

Key components or key attributes are the member of the key structure. A key may have any number of key components, but the maximum length for a key is limited to 512 bytes.

Key components must be defined as attributes in the class or view the key belongs to.

### Array components

When defining a key on an array attribute, as many keys as array elements exist are created for each instance. This allows accessing an instance by more than one key.

A key definition may contain only one array component. When containing an array component, the key must not contain a generic key component.

## Generic key components

One key component in a key definition may refer to a generic attribute. Defining generic key components differs from array keys, because for each type of the generic attribute a separate index will be created when the key is referenced in an index definition.

This allows e.g. defining language dependent indexes in case of having language depending attributes (generic attributes), where the language defines the generic type of the attribute.

When a key contains a generic key component, it must not contain an array component.

## Identity key

A special key is the `__IDENTITY` key, which refers to the implicitly defined `__IDENTITY` attribute, which exists for each persistent data type. The `__IDENTITY` key must get the name `__IDENTITY` and does not have key components.

```
CLASS Person
(
  KEY __IDENTITY;
  ...
)
```

You may also define the `__IDENTITY` key as identifying key, which makes sense, when there is no identifying set of attributes available in the class definition. Identity keys cannot be defined for views.

## Component options

Key components can be extended by several options, which determine the way of comparing keys.

**IGNORE\_CASE** – The option indicates, that comparing keys the difference between upper and lower case letters for text components will be ignored.

```
CLASS Person
(
  KEY sk_Name (IGNORE_CASE DESCENDING name,first_name);
  ...
)
```

**DESCENDING** – The option indicates, that the key components are ordered in descending order when being referenced in an index definition. Descending order applies only on key components, which have been marked as such. Other key components are stored in ascending order in the index.

Even though it seems, that key component options are index attributes rather than key attributes, since those influence the sort order in the index, it has been decided to define them as component attributes. This has the advantage, that those options can be defined separately for each key component.

## Key references

You may refer to keys in the application by referring to the key name and one of its component names.

```
SET<Person>    &persons = Persons;
...
persons.get('ID0033').
persons.sk_name.name
```

When comparing key components, the `IGNORE_CASE` option is in affect, which is not the case, when accessing the key component attribute (`name`) directly, instead.

## ORDERED\_BY

Typically, keys are referenced in index definitions for collections (extents, references or relationships). Indexes are used for defining persistent or temporary sort orders for a collection.

Indexes are defined by using the `ORDER BY` option.

```
CLASS Person
(
  KEY {
    IDENT_KEY ik_pid(pid);
    sk_Name (name,first_name);
  };
  EXTENT Persons MULTIPLE_KEY OWNER
    ORDERED_BY (ik_pid UNIQUE SUPRESS_EMPTY, sk_Name);
)
```

For each collection any number of indexes can be defined after the `ORDERED_BY` keyword. Indexes are defined by referring to the key name as defined in the view or extent definition the extent, reference or relationship is based on. Index definitions must be enclosed in parenthesis and separated by comma. Eck key reference in an index definition can be preceded by a number of options:

**UNIQUE** – The option indicates that key values in the index are unique. An attempt to add the same key value twice into the index will result in an error.

**SUPRESS\_EMPTY** – This option excludes empty key values from the index, i.e. when adding an instance to a collection, it will not be added to the index, when all key components contain empty values. This option should never be set for identifying keys.

**TEMPORARY** – A temporary index is created at runtime and is not stored to the database. Temporary indexes should not be defined for the identifying key.

The current OSI version allows defining one index per key for a collection, i.e. you cannot define several indexes for the same key with different index options for one collection.

# 6 Variables

Variables are symbols in an OSI function, which represent values when executing the function. Depending on the context, in which a variable has been defined, OSI distinguishes between:

- **Member variables**  
Member variables (properties) are defined in the context of complex data type definitions. This is explained in detail in chapter “Data types”. Member variables can be of any of the variable types listed in the variable type hierarchy below (attribute, reference, relationship, inheritance, key). In some cases, also methods are considered as member variables.
- **Parameter variables**  
Parameters are variables that are passed to other methods (functions). Parameters are declared in the function header. Details about parameter definitions are described in chapter “Function Parameters”.
- **Local variable**  
Local variables are member definitions in the variable section of an OSI function. Local variables are valid in the context of a function and can be inherited to local sub-functions. Details about how to define local variables are described in chapter “Variables Section”.
- **Global variable**  
Global variables are member definitions, which are defined outside an OSI function or which are preceded by the `GLOBAL` keyword. Details about how to define global variables are described in chapter “Global variables”.
- **Database variables**  
Database variables are variables defined in the database. Usually, database variables are global database and view extents defined in the dictionary. Details about using database variables are described in chapter “Database variables”. Also enumerations are sometimes considered as database variables.
- **Options (system variables)**  
Options or system variables are variables, which can be set in an option file or in the application. Options are thread-global, i.e. the same option may have different values in different threads.



- **Self variable**

The self variable (`self`) refers to the object instance or collection that was calling the function.

- **Extension variable**

Extension variables are based on extension property definitions and may be referenced for any extendable data type. Extendable data types are data types that inherit from `__OBJECT` or that contain a reference property `__EXTENSIONS`. Extension variables are potentially available for each extendable data type, regardless whether those have been assigned to the object instance or not.

Variables behave in the same way in general, but they differ in the way they are defined.

**Variable type hierarchy**

Besides simple member variables, variables can be defined as more specific variables as shown in the subsequent variable type hierarchy::

- Variable
  - Attribute
  - Reference
    - Relationship
      - Inheritance/extension
  - Key

Attributes, references and keys ARE specific variables that can be defined in the context of a complex data type (class, view, interface). Relationships are specific references and inheritance or extends specifications are specific relationships.

OSI specific is the fact, that inheritance/extends (base classes) are considered as variables and can be accessed as such, since a variable name can be assigned to each inheritance or extends specification.

**Variable operations**

A number of operations can be used in connection with variables. Variable operations and their semantics are described in chapter "Built-in operations".

Besides built-in operations, OSI provides more than 200 access functions for variables. Most of them apply to persistent variables, for which data is stored in the database. Access functions for variables are described in chapter "Built-in class functions".

**Cursor function** Variable semantics in OSI differ from traditional programming languages, since OSI variables have got cursor functionality, i.e. an OSI variable is considered as collection of instances of a given type (variable type). One instance can be selected for a variable, which is considered as the current value of the variable.

Variables, which do not really represent a collection, because their dimension is 1, i.e. they contain exactly one instance, are considered as collections with one instance.

The cursor function allows selecting one instance as current variable instance. Many operations operate on the current instance rather than on the whole collection. There are, however, functions, which operate on the collection as `copyCollection` or `lockCollection`.

**Selecting an instance** Selecting an instance for a variable is usually done by the selector operator `()`, which is nearly equivalent to the `get` function call.

Using selector operators will reset filter conditions which might have been set before or inherited from the variable origin.

```
person(0);           // select first person instance
person.get(0);       // same as above
person('ID0033');    // select person with pid ID0033
person.get('ID0033'); // same as above
```

Selecting an instance for a variable is possible by position or by key, when the variable represents an ordered collection. When the order key has more than one component, component values are separated by `|`.

```
person('Miller|Paul');
```

The difference between using the `get` function and the selector operator is, that calling the `get` function will change the selection in the `person` variable, while the selector does not.

```
person('ID0033');
Message(person.name); // no person selected,
                       // NULL value exception
person.get('ID0033'); // selects an instance in person
Message(person.name); // prints: Miller
```

After selecting an instance in the variable of complex data type, you can access member variables of the selected object instance or perform instance operations.

```
person('ID0033'); // select person with pid ID0033
person.name = 'Miller'; // assign a name to the selected person
// or in one statement:
person('ID0033').name = 'Miller';
```

When defining simple application variables as in the example below, selecting an instance is not necessary, but would also not cause an error, because the cursor functionality is supported for any type of variable.

```
string name = 'Miller|Paul';
person(name);
```

Instead of a constant value, one may also pass a variable or expression as locator. The variable or expression will be evaluated in the scope of the function defining the path. When passing an expression, the expression is evaluated and the result is passed as locator to the collection.

```
STRING app_name;
...
app_name = 'Sample'; // same as: app_name(0) = 'Sample';
```

Applying instance operations on variables, which do not have a selected instance will cause an exception.

**Built-in operations** Local variables support built-in operations and built-in class functions as described in the corresponding chapters. Moreover, local variables support functions defined for the data type of the local variable.

**Class methods** When the data type is a complex data type (class) defined by the application, any number of methods can be defined for the data type, which can be called with the local variable. Moreover, interface functions can be called for complex data type variables, which provide an OSI function interface.

### **Transient and persistent variables**

Depending on the way, variable values are stored, variables can be divided into two groups:

- **Transient variables**  
Transient variables are variables, for which values are stored in memory and which are available during the life time of a process, only.
- **Persistent variables**  
Persistent variables are variables, for which values are stored in the database and which survive the process.

Within OSI transient and persistent variables are handled in the same way, i.e. they do not differ in the way they operate. The main difference is that values for persistent variables are read from the database and stored automatically after being modified. Thus, programming with persistent variables is completely the same as programming with transient variables.

Transient variables

Transient variables are available during the life time of a process. The lifetime of a variable in a process depends on the type of variable definition.

Global variables are accessible until the end of the process. Local variables and parameter variables are accessible as long as the function runs. The accessibility of member variables depends on the type of variable the member variable belongs to.

Persistent variables

Persistent variables are variables, that are usually stored in the database and thus, survive the process, that has created or updated those variables.

Referring to persistent variables requires database access. Since database access may fail for different reasons (access rights, no instance available, etc.), referring to persistent variables may cause an exception. Exceptions are always fired, when accessing persistent variables fails. Exceptions fired can be handled in the same function or in any other function in the calling hierarchy.

Updating persistent variables will cause updating the database. This is done automatically, when another instance is selected for a variable and does not require additional actions. The time point, however, when updated data is really stored to the database, can be controlled by the application calling the `save` function explicitly.

Persistent variables are usually member of complex data types. Persistent variables are stored, when the instance, they are part of is stored to the database. This is done automatically, when another instance is selected for the variable, or when the explicit `save` function is called.

## 6.1 Database Variables

Database variables are variables, which are part of the database definition and which are stored in the database. Typically, all extents defined in the dictionary are database variables.

Since database extents are the entry point for accessing the database, one has to define an extent also for defining a single database variable.

Instead of persistent extents, view extents can be referred to as database variables as well. View definitions (view type) are not database variables but can be referred to as method.

### Database extents

Since database variables are part of the database, they have been declared as extent definitions when defining the class that defines the instance structure for the database. Hence, database variables need not to be declared, when being accessed in an OSI function.

```
Persons('ID0033').name = 'Miller';
```

The example above just refers to the database variable `Persons` defined as extent in the database dictionary.

Database variables exist only in the context of an access path (`Persons('ID0033').name`). Several references to the same database variable in different access paths are considered as different variables, i.e. they will not have the same cursor position.

```
Persons('ID0033'); // select instance  
Persons.name = 'Miller'; // other variable, exception
```

In the example above, the database variable in the second line is different from the database variable in the first line, even though they refer to the same collection (extent). Since the database variable in the second line is not positioned (no instance selected), the second statement will terminate with an exception.

To handle such situations, database variables should be assigned to local or global variables:

```
Person &pers_coll[] = Persons; // assign DB variable  
...  
pers_coll('ID0033');  
pers_coll.name = 'Miller';
```

After assigning the database variable `Persons` to the local `pers_coll` variable, the function performs well, since the local variable `pers_coll` is positioned in line three of the example, and the same positioned variable is used in line four for assigning the name.

## View Extent

Besides database extent definitions you may define view extents in the database dictionary. A view extent is an application of a view type to one or more data collections.

View extents are considered as database variables as well and can be referenced in OSI functions.

As well as database extent variables, view extent variables are defined in the scope of an access path. Depending on the way of referring to view variables, view extents are created at once (offline view) or view instances are provided on demand (online view).

```
Employment &pers_comp[] = Employments; // online view
Employment pers_comp[] = Employments; // offline view
```

Assigning a transient view extent by reference creates an online view, i.e. view instances are displayed with the state they have got when the view instance was requested. When assigning the transient view extent by value or when assigning a persistent view extent, the extent will be created when not yet existing and the extent variable is assigned to the local variable like a database extent variable.

## Offline views

Creating an offline view for a transient view extent causes ODABA to create a temporary extent in a temporary database area. Temporary extents can be accessed as long as the process is running.

Creating a new cursor for an offline view can be done by assigning the view variable:

```
Employment pers_comp[] = Employments; // offline view
Employment copy[] = pers_comp; // offline view copy
```

In this case, the view is not re-created, but a new cursor is created for the temporary view extent.

## 6.2 Global Variables

Global variables are variables, which can be accessed in any scope of the application. Global variables are transient variables and available within a process, only. Global variables can be defined in function definitions and outside any scope, but not within class or view definitions.

Variables defined in an OSI script file outside any class, view or function scope, are global variables by default.

```
// global variables
STRING    application_name = 'Sample';
FileHandle docFile;
```

It is suggested to define global variables outside any scope, but it is not necessary. You may also define global variables within an OSI function:

```
BOOL OpenFile()
{
VARIABLES
    GLOBAL    FileHandle docFile;
    STRING    filename = 'c:/ODABA/Sample/Sample.txt';
    BOOL      cond = TRUE;
PROCESS
    if ( docFile.IsOpened() )
        Message('Warning: File already opened - reopened');
    if ( cond = docFile.Open(filename) )
        Message('Error : File '+filename+'could not be opened');
FINAL
    return(cond);
};
```

Global variables are created when being defined the first time. Redefinitions for global variables are ignored and do not cause an error. Thus, redefining a global variable with different specification will be accepted but will not change the variable specification.

When a global variable has already being defined, it needs not to be redefined in the functions, which refer to the global variable. When defining a local variable in a function with the same name as a global variable, the local variable will overwrite the global variable in the function.

Global variables can be base type variables as well variables with complex data type.

## 6.3 Self variable and execute operator

Self (`self`) is a variable, which is available in any non-static function. Self refers to the object instance or collection, which was calling the function.

Self represents exactly the current state of the calling object. Thus, changing the selection in a collection object are reflected in the self variable.

### Usage

Usually, the self variable is not necessary, since object or built-in methods can be called directly and need not to be prefixed by the self operator. When, however, variables have been defined with the same name as e.g. built-in functions, the self variable allows to distinguish between variable reference and function call.

```
COLLECTION STRING Person::ChildrenList()
{
VARIABLES
  Bool      first = true;
  STRING    childrenList;
PROCESS
  If ( self.first ) { // selects first instance in collection
    while ( located ) { // same as self->located
      childrenList += name;
      if ( !first ) // refers to local variable
        childrenList += ',';
      first = false;
      next;
    }
  }
FINAL
  return(childrenList);
};
```

The example above could be written a little bit simpler, but it shows the different ways of referring to first as function and as variable. .



## 6.4 Lookup priorities

When referring to a variable in an OSI function, variables or symbols (names) are searched in the following order:

### Local scope

- Local variables
- Parameter variables
- Member variables
  - Class properties
  - Meta-attributes
  - Namespace database variables
    - Local extents
    - Local enumerations
  - Property extensions
  - Methods
    - Actions
    - OSI functions
    - CPP functions
  - Base type variables (local scope)
- Built-in functions (odaba/odabagui interface)

### Global scope

- Global variables
- Database variables
  - Global enumerations
  - Global extents
- Global OSI functions
- OSI Constructor functions
- System functions

When the current class is defined in a namespace hierarchy, extents and enumerations defined in upper namespaces can be accessed.

When variables are ambiguous, the scope operator can be used to refer to lower priority variables.

```
// Message is defined as database extent
// to output a system message requires now:
SystemClass::Message('Hello ...');

// When a function locate has been implemented in the Person
// class, the property handle function locate can be called as:
person.Property::locate(...);
```

## Execute operator

Since local variables, parameters and class members are resolved prior to methods, problems occur when those variables have got the same name as built-in or class interface functions. In order to distinguish between variables and methods, the execute operator (->) can be used..

```
... Person::fragment()  
{  
    Message(children(0).first); // first attribute for first child  
    Message(children->first.first); // same as above  
};
```

The execute operator (->) always requires a method. Thus, using the execute operator allows referring to a function, also, when a variable with the same name is available.

# 7 Functions

OSI functions are based on a syntax similar to Java. OSI supports three types of functions:

- **Global functions**  
Global functions are defined outside of any class definition and do not belong to a class. Thus, global functions behave like static functions.
- **Class functions**  
Class functions are defined in the scope of a class or must have a scope operator, which defines the class the function belongs to. Class functions can be defined as static, instance or collection functions.
- **Inline functions**  
Inline functions are functions, which are defined in another function. Inline functions are considered as class functions of the class of the calling object. Inline functions do not have parameters, i.e. you cannot pass parameter values to inline functions. Instead, inline functions may refer to variables defined in the calling function.

All functions may contain different sections for defining variables, function code, error handling and final processing.

## General format

OSI functions have got a standard format, which is independent on the function type.

```
[ function header ]  
{  
[ VARIABLES  
  Variable definitions ]  
[ PROCESS ]  
  statements  
[ ON_ERROR  
  statements ]  
[ FINAL  
  statements ]  
};
```

### Function header

The function header defines the type of the function and the parameters. The function header must be defined for global and class functions. It must not be defined for inline functions.

Variable section	The variable section contains definitions for local variables, i.e. variables, which are defined in the scope of the function. The section must be preceded by the <code>VARIABLE</code> keyword, which is not necessary, when no local variables are defined for the function.
Process section	The process section contains the statements for the function. The <code>PROCESS</code> keyword can be omitted, when no variables are defined for the function.
Error section	The error section is a block of statements, which is executed in case of an error or exception. It must start with the <code>ON_ERROR</code> keyword, when being defined. It can be omitted, when no specific error handling is required.
Final section	The final section is executed, when leaving the process or error section ( <code>leave</code> ) or when an error has been signaled ( <code>error</code> ). It is not called in case of an unhandled exception. Exceptions are passed to the next <code>ON_ERROR</code> block in the calling hierarchy.

## 7.1 Function Header

The function header defines the type of the function and the parameters.

```
[FUNCTION] [options] type expr_name( [parameters] )
```

The function header may start with the `FUNCTION` keyword to distinguish between functions and other method types (e.g. C++ or JAVA). When running OSI, OSI functions are the only methods that can be defined. In this case, the `FUNCTION` keyword can be omitted. It must be defined, however, when running the LoadSchema utility, which supports different types of function definitions.

## 7.1.1 Function Options

Options are provided for defining the function type, access privileges and other function properties

### Function types

As in other object-oriented languages, most OSI functions require a calling object, which is the base for executing the function. You may also define **STATIC** functions, which do not require any calling object.

In addition, you may also define functions, with a collection as calling object. The difference must be defined in for the function, since variables represent collections as well as instances. Thus, it makes a difference, whether a function is called for the instance selected for the variable or for the collection represented by the variable.

Several options are provided for defining different types of calling object for the function. There are three ways, a function can work:

- Calling object is a collection
- Calling object is an instance (default)
- No calling object required

**STATIC** – indicates, that no object instance or collection is required for executing the function.

### Static function

Static functions are functions that do not require a calling object. Static functions can be called from global functions or functions belonging to other classes by using the scope operator. Within static functions, member variables defined for the complex data type (e.g. class) the function belongs to are not available. Static functions must be marked as such, when being defined as method of complex data types.

```
STATIC INT Person::Count()  
{  
    return ( Persons.count );  
};
```

All functions not defined as method for a complex data type are static functions by default. Functions defined as methods for a complex data type must explicitly be declared as static using the **STATIC** keyword.

In the context of a class method, static functions of the same class can be called without any prefix.

```

STATIC INT Person::Example()
{
VARIABLES
  INT      count = Count; // calls Person::Count
  ...
};

```

Outside the scope of the complex data type defining the static function it can be called by using scoped names or passing a calling object:

```

count = Person::Count; // scoped name
count = Persons.Count; // passing calling object

```

Passing a calling object, the calling object is used only for determining the complex data type the method belongs to. No data from the calling object is accessible in the static method.

#### Collection function

Collection functions operate on the collection rather than on a single (selected) instance. Hence, collection functions do not require a selected instance.

Defining a collection function requires the **COLLECTION** keyword in the function header:

```

COLLECTION BOOL Person::Count()
{
  return count;
}

```

You may call a collection function with a single instance or a collection without any problem, since a single instance is considered as collection with one instance in this case.

```

Persons.Count; // returns the number of persons
Persons('P2-10').Count; // returns 1 when person P2-10 exists

```

Since the result of `Persons('P2-10')` is an instance and thus, the `Count` function returns 1 in this case.

When applying collection functions on a collection with a selected instance, the function may change the selection. On the other hand, the called function may use this information. When passing a selected `Persons` collection to the collection function `Print`, the function might print all the persons beginning with the selected instance.

## Instance function

Calling an instance function requires a variable with a selected instance. Variables containing a single instance, only, will automatically select the only instance if not yet done.

Instance is the default. There is no keyword for instance and instance is assumed when neither `STATIC` nor `COLLECTION` have been defined.

```
Persons('P2-10').Print; // calls print for person P2-10
Persons().Print;      // calls print for each persons
```

The first example selects a person in the `Persons` collection before calling `Print`. The second example iterates through the collection and calls `Print` for each person in the extent.

A call of `Persons.Print`, however, will work properly only, when a person has been selected in the `Persons` extent before.

Instance functions may change member variables of the selected instance for the variable passed as calling object to the function, but they cannot change the selection in the calling object.

## Virtual function

Inheritance is used similar to other object-oriented environments. Functions defined in base classes, are inherited from the specialized classes.

A virtual function in a specialized class may overload a function defined in a base class (generalization).

When defining a virtual function, OSI checks at run-time, whether there is an overloaded function in the specialized class, which applies to the currently selected instance.

In case of weak-typed collections, OSI supports dynamic binding for functions that are defined as virtual in the base class of the collection. Thus, specialized instances will run specialized functions.

Considering a `Persons` collection, which contains different specialized instances for persons like `Student`, `Retired`, and `Employee`, one may implement a virtual `Print` function in the `Person` class.

```
VIRTUAL BOOL Person::Print()
...
```



Re-implementing the `Print` function for `Student`, `Retired`, and `Employee` will print each person instance according to the `Print` function implemented in the specialized class when calling `Persons().Print`.

`VIRTUAL` does not make sense for collection functions. You may define it, but it will not have any effect, since OSI does support class inheritance, but not collection inheritance.

### **Access privileges**

Functions can be defined with different access privileges:

**PUBLIC** – Function can be called from anywhere

**PROTECTED** – Function can be called from other class functions, when the calling class inherits from the class that implements the function.

**PRIVATE** – Functions can be called with calling objects of class only, which implemented the function.

OSI allows defining access privileges, but this feature is not yet supported, i.e. the definition will not have any impact to the execution of a function. The option should, however, be used with care since it is planned to support access privileges in future OSI versions,

### **One way**

**ONEWAY** - indicates, that the function does not change the calling object. This option has been added for compatibility with ODMG suggestions, but has no effect in OSI for ODABA 9.

## 7.1.2 Type of Returned Value

The type defines the type of the return value. The return value can be any of the supported types, i.e. an elementary type (CHAR, BOOL etc.), a complex data type (Person, Car), a template type (ARRAY<Person>) or any other data type.

Return values might be passed directly or by reference. Returning an instance by reference returns a “pointer” to the returned instances, i.e. updating the returned value has an impact on the “original” instance as well.

```
COLLECTION &Person Person::FindTop()  
...
```

When a reference to an instance in a collection is returned, the instance can be accessed, but the selection in the collection cannot be changed. On the other hand, changing the selection in the original will change the instance in the reference as well.

You may also pass collections by reference. In this case, the returned (referenced) collection and its original using the same cursor (i.e. referring to the same selected instance), and the selection can be changed in the reference and in the original collection.

### 7.1.3 Function Parameters

Parameters are variables, which are passed to a function when calling it for execution. Parameter variables are defined in function header. When calling a function, values are assigned to parameter variables and passed to the called function.

```
BOOL Person::CreatePersons (INT number)
{
    ...
};

bool main()
{
    if ( Persons.CreatePersons (10) )
        ...
}
```

Parameter values are automatically converted when required (see “Data conversion”). The type of assignment (by value or by reference) when assigning operands to parameters depends on the type of parameter definition. Parameters defined as reference variables are assigned by reference. Parameters defined as value variables are assigned as value variables.

#### **Parameter declaration**

Parameters are declared in the function header. OSI does not support function declarations as in C++. The only place declaring parameter variables is the function definition.

```
type_spec expr_name( [params_decl] )
```

Each parameter in the parameter declaration list (params\_decl) is defined as:

```
[param_attribute] simple_type_spec declarator
```

In contrast to local variables, parameter declarations allow only one declarator per parameter. Parameter declarations are separated by comma.

## Default value

Each parameter variable may get a default (initial) value. As initial values constants can be defined, but also operands, which are valid in the context of the defined function. Operands may refer to global and database variables, to member variables, when the function is an instance function, but not to local variables defined in the function or to other parameter variables.

The initial value is determined for missing parameters when calling the function based on the calling object instance.

When the function is a collection function, initial parameter expressions should refer to constant values or global or database variables, only.

Default values can be assigned to parameters by defining an initial value in the parameter declaration:

```
BOOL Person::CreatePersons(INT number = 10)
```

Initial values are assigned to the parameter, when no operand is passed to the parameter declared in the function header when calling the function.

## In and out parameters

According to ODMG suggestions, parameter can be declared as `IN`, `OUT` or `INOUT` parameters (`param_attribute`). This option does not have an impact in OSI functions, since only parameters declared reference variables may function as `OUT` or `INOUT` parameters. OSI does not check, whether the variable type (by value or by reference) is consistent with the in/out specification for the parameter.

## 7.2 Function Body

The function body consists of different sections, which are optional, except the `PROCESS` section. The function body is enclosed in `{ ... }`.

```
{
  [ VARIABLES
    Variable definitions ]
  [ PROCESS ]
    statements
  [ ON_ERROR
    statements ]
  [ FINAL
    statements ]
};
```

- **Variable section**

The variable section contains definitions for local variables, i.e. variables, which are defined in the scope of the function. The section must be preceded by the `VARIABLE` keyword, which is not necessary, when no local variables are defined for the function.

- **Process section**

The process section contains the statements for the function. The `PROCESS` keyword can be omitted, when no variables are defined for the function.

- **Error section**

The error section is a block of statements, which is executed in case of an error or exception. It must start with the `ON_ERROR` keyword, when being defined. It can be omitted, when no specific error handling is required.

- **Final section**

The final section is executed, when leaving the process or error section (`leave`) or when an error has been signaled (`error`). It is not called in case of an unhandled exception. Exceptions are passed to the next `ON_ERROR` block in the calling hierarchy.

Each section except the `VARIABLES` section consists of a sequence of statements. OSI does not support labels and `GOTO` instructions, but these sections help to control the process flow and error handling.

## 7.2.1 Variable Definitions

Local variables are variables, which are defined in the scope of a function. OSI functions do have a separate section for defining variables. Variables can be defined only in the variable section.

```
bool main(String applname='Test')
{
VARIABLES
  GLOBAL STRING application_name;
  STRING(40)  applname;
  BOOL      cond = true;
PROCESS
  ...
}
```

The end of the variable section is indicated by the `PROCESS` section label.

The variable section may contain global and local variable definitions. The definition of global variables is described in detail in chapter “global Variables”. Here, the definition of local variables is described.

### Variable definition

Local variables are always defined as member variables, i.e. you cannot define a relationship or reference as local variable.

```
domain_type declarators ';' ;'
```

Each variable definition consists of a type specification and one or more declarators. OSI does not support local definitions for complex data types of enumerations, i.e. all types referenced in local variable definitions must be defined in the dictionary or as explicit data type definitions in the script file. Details for defining member variables are described in chapter “Members”.

Local variables can be initialized by assigning a value to the variable.

```
INT(10)    count = parm1;
INT(10)    double_count = count*2;
```

Operands assigned to the variable may refer to parameter variables, to global, database and member variables and to local variables defined before.

Variables can be defined as value or reference variables and initial values can be assigned to variables by reference or by value. Details for variable assignments are described in topic “Built-in Operations/Assignment operations”.

#### Operation path variables

An operation path variable allows assigning an operation path (or view) to a variable. When assigning an operation path as initialize value to a variable, the operation path is executed immediately and the result is assigned to the variable.

```
SET<Person> grand_children =  
    Persons().children().where(age < 18);
```

For property paths, the result is the last property referenced in the path.

```
SET<Person> grand_children = Persons().children;
```

In the example above, this is the `children` collection. When assigning property paths, iteration operands will be ignored. In order to refer to operation or property paths as such, reference variables should be defined.

#### Reference variables

Reference variables are defined by a preceding `&` in front of the variable name. Reference variables can be initialized by reference assignment, only. Regardless, whether reference assignment is requested explicitly or not, initializing reference variables always uses assignment by reference.

```
SET<Person> &grand_children &= Persons().children;  
SET<Person> &grand_children1 =  
    Persons().children().where(age < 18);
```

When assigning an initial value to a reference variable, the initialize expression (right hand operand) is opened as access handle and assigned to the reference variable. This allows iterating through the operation, which is different from iterating through the result of an operation as in case of non reference variables.

## 7.2.2 Processing

The processing section of a function contains the statements for performing the function operation. Processing sections might be rather complex but may also refer just to a single statement.

Simple functions just contain the processing section need not to introduce the processing section by the `PROCESS` keyword.

```
INT Person::Count()
{
    return count;
};
```

Functions containing one or more variable definitions need to introduce the processing section by the `PROCESS` keyword, which follows the list of variable definitions.

```
BOOL PrintPersons()
{
    VARIABLES
        GLOBAL    FileHandle docFile;

    PROCESS
        ... processing statements
}
```

When calling a function, the processing section is entered after initializing parameters (default values) and variables. Several syntax operations allow altering and controlling the process flow within the processing section (see also chapter “Process Flow Operations”).

Processing is the `PROCESS` section is terminated, after the last statement has been executed or when termination is caused by a process flow operation.



## 7.2.3 Error handling

The `ON_ERROR` block supports error handling. OSI differs between local and global errors.

```
BOOL Example()  
{  
  ...  
  if ( ... ) error(99)  
  ...  
  ON_ERROR  
    return FALSE;  
  FINAL  
    return TRUE;  
}
```

### Local Errors

Local errors are errors, which can be handled within a function, only. Local errors are indicated by using the `error` function ("Process Flow Operations"). The `error` function causes a break in the processing and passed control to the `ON_ERROR` block defined in the function.

When using the `error` function in a function, one should always define an `ON_ERROR` section for handling the error. When not defining an `ON_ERROR` section and calling the `error` function in the function, processing continues with the final block, when being defined, or leaves the function.

### Exceptions

Exceptions or global errors are errors, which are created by OSI or explicitly by calling the `exception` process flow function. An exception passes control the next `ON_ERROR` block in the calling hierarchy. When no `ON_ERROR` block is defined in the calling hierarchy, processing terminates abnormally.

When reaching the `ON_ERROR` block, the `_LastException` variable contains the exception reason.

```
BOOL Example()  
{  
  ...  
  Message( get(0).value("name") );  
  ...  
  ON_ERROR  
    Message( "error " + (string) _LastException.errorCode +  
            ": " + _LastException.errorReason );  
  FINAL  
}
```

The last can be checked referring to `_LastException` properties provided by the `odaba::Exception` interface.

The default exception handling can be changed by using the `exceptions` directive or setting the OSI `exceptions` option to a valid exception handling mode before starting the application.

```
BOOL Example()  
{  
  ...  
  #exceptions accept;  
  X = a + undefined;  
  #exceptionS;  
  ...  
}
```

Changing the exception handling state will be in effect until it is changed again or until leaving the function, that has changed the exception handling. You may also restore the previous settings by using the `exceptions` directive without any parameter.

#### Handling types

In the example above, the red line is supposed to through an exception. When not using the `exceptions` directive, the function terminates and tries to locate the next higher `ON_ERROR` block. To avoid this, you may change the exception handling as described below:

**accept:** When, setting exception handling to accept, the line throwing the exception is executed by assuming NULL values or the operand(s) causing the exception.

**ignore:** Setting exception handling to ignore, causes the statement throwing the exception to be ignored. Processing continues with the statement following.

**exception:** is the default setting, which causes the function to continue with the nearest `ON_ERROR` block in the calling hierarchy.

#### Restoring previous value

For restoring the exception handling state as it was before changing it, you may call the `exceptions` directive without any parameter.

#### `exceptions` option

In order to change the default for exception handling when running OSI functions, the OSI `exceptions` option or environment variable might be set to one of the valid values. Invalid settings will be ignored.

## Null value handling

Special exceptions are “null value exceptions”, which are fired, when operating with variables containing no value. Many operations require a selected value for a variable and cannot work properly, when no value is selected for the variable.

Statements containing “null-values” cause an exception by default. The specific handling of null value exceptions may, however, be controlled by the `#nullvalue` directive or by setting the OSI `nullvalue` option.

```
BOOL Example()
{
  ...
  #nullvalue accept;
  DocFile.Out("The name is: %s",Persons("Miller").name);
  #nullvalue exception;
  ...
  ON_ERROR
    return FALSE;
  FINAL
    return TRUE;
}
```

### Handling types

**exception** (default): In the example above, the file output function will not work properly, when no person Miller can be selected. When not using the `#nullvalue` directive, the statement causes a null-value exception, because evaluating the parameter fails. Control is passed to the next `ON_ERROR` block in the calling hierarchy (as long as exception handling does not define different behavior).

**accept**: When, however, setting the “null value handling” to accept, the line is executed and a `#nullvalue` is passed to the function. In this case, the text “The name is: “ is written to the output file.

**ignore**: When setting null value handling to ignore, instead, the line will not be executed and processing continues with the next statement.

```
...
#nullvalue ignore;
DocFile.Out("The name is: %s",Persons("Miller").name);
...
```

In order to reactivate the default handling, you may set the `#nullvalue` directive to “exception” after executing critical statement(s).

Calling functions

When calling sub-functions, the current state for `NULLVALUE` handling is passed to the sub-function. After returning, the `NULLVALUE` handling works as being defined in the current function regardless on the state set in any function called from the current function.

In order to reset the `NULLVALUE` handling to the current state of the calling function, the `NULLVALUE` directive can be called without parameters.

```
...
#nullvalue accept;
  DocFile.Out("The name is: %s",Persons("Miller").name);
#nullvalue;
...
```

OSInullvalue option

In order to change the default for `NULLVALUE` handling when running OSI functions, the `OSInullvalue` option or environment variable might be set to one of the valid values. Invalid settings will be ignored.

## 7.2.4 Final Section

The `FINAL` section contains a number of statements, which are executed before leaving the function. Usually, the `FINAL` section provides the return value.

When being defined, the `FINAL` section is called, after the last statement in the `PROCESS` section has been processed or when the `PROCESS` section has been terminated with an `leave` or `break` operation (“Process Flow Operations”). The `FINAL` section is also called in case of an error after processing the statements in the `ON_ERROR` block.

The final block is not processed, when leaving the `PROCESS` section with the `return` function (Process Flow Operations).

When a return value has been defined for the function, an operand can be passed to the `return` function, which is returned as return value from the function. When not calling the `return` operation or when not passing an operand to the `return` function, the value from the last statement executed is returned.

## 7.3 Constructor

OSI supports constructor functions in order to initialize complex data types. Constructor functions are functions that get the same name as the data type and are used to initialize an instance with a complex data type. Constructor functions can be used for initializing local variables in a OSI function.

```
Color    my_color(100,100,100); // initialize local variable
```

One might call a constructor also with an object instance in order to initialize the object in the PROCESS section of a function.

```
VARIABLES
  Color    my_color;           // create variable
PROCESS
  my_color.Color(100,100,100); // initialize variable
```

## 7.4 Statements

Statements are basic operation units of a function. Each function is a sequence of statements, which are executed as they are defined.

## 7.5 Global Functions

An OSI application must contain at least one global function, which is used as entry point for the application. By default, the entry point for the application is `main`, but you may use other names or several entry points to the application, which must be passed to the OSI interpreter when calling the application.

```
bool main (STRING applname)
{
    ...
}
```

In contrast to class functions, global functions are not preceded by the `FUNCTION` keyword.

From within a global function you can call static or non-static class functions, only, but no other global functions. This means, that all functions, except the entry point functions, must be defined as class functions. General purpose functions can be provided in a “Common” class as static functions.

In the scope of a global function one has only access to local and global variables, extents and static class functions, i.e. each access path in a global function must begin with a global or local variable, an extent name or a class name (as scope operator).



## 7.6 Class Functions

## 7.7 Local Functions

## 8 Operation Reference

This section provides a list of built-in operations and their functionality. Built-in operations are operations, which are provided as syntax elements, but also functions supported for ODABA collection template classes and other built-types.

An OSI built-in operation is a statement, which is expressed as:

- Syntax function
- Built-in class function
- System function

## 8.1 Syntax Functions

Syntax functions are functions, which are part of the OSI syntax. Syntax functions are supported as function operations, process flow operations and query operations.

## 8.1.1 Process Flow Operations

Process flow operations allow controlling the execution sequence of statements. Process flow operations support conditional processing and specific jump operations (process control).

### Conditional processing

Conditional processing provides some syntax function, which can process a statement or a block of statements after checking certain conditions.

`if` | `IF`

An `if`-block defines a conditional execution of statements. When the condition returns true, the statement or block following the condition is executed. If not, the else-statement or block is executed, when being defined.

```
if ( a > b ) c = a-b;
else       c = b-a;
```

`switch` |  
`SWITCH`

A defines a complex conditional execution of statements. Each is compared with the `switch`-operand. The statements for the first `CASE`-operand that returns true for the comparison, are executed until the next `break` or `leave` statement,

```
switch ( sex )
{
  case "Female" : text = "Mrs.";      break;
  case "Male"   : ;
  default      : text = "Mr.";
}
```

In contrast to C++/Java, `CASE`-operands and `switch`-operands may contain operand expressions instead of constants. Moreover, empty `CASE`-statements require a semicolon, as well as the `break` statement.

`while` |  
`WHILE`

A `while`-block defines a conditional loop. As long as the condition returns true, the statement or the statements in the block are executed.

```
While ( Persons.next )
  Total = total + Persons.income;
```

for | FOR

A `for` block defines a loop initialization in the first multiple operand statement (`init_operands`). Initialization is executed once each time, when the `for` block is entered.

The next multiple operand statement may contain one or more conditions, which are combined by logical `AND`. When all conditions are true, the statement or block defined after the `for` statement is executed. When at least one of the conditions returns false, the `for` loop terminates.

The final multiple operand list will be executed after each loop iteration, e.g. for increasing the loop count.

```
for ( i=0, j=0; i<10, j<10; i=i+1, j=j+1 )
    c = c + i*j;
```

**Process control**

Process control statements allow changing the process sequence of statements. OSI does not support `GOTO` statements. Instead, additional process control statements have been added, which allow handling errors and exceptions.

All process control statements require a semicolon at the end.

continue |  
CONTINUE

The `continue` operand causes the function to start the next iteration of a loop. The `continue` operation makes sense only within a `while` or `for` block. Within a `for` block, the `continue` operation will continue with the final operand list.

```
for ( i=0, j=0; i<10, j<10; i=i+1, j=j+1 )
{
    if ( j==7 )          continue; // next iteration
    c = c + i*j;
}
```

break |  
BREAK

The `break`-statement terminates the processing of statements in a block and goes to the end of the block. The block end is the next `}`, which closes the block of statements, where the `break` has been called.

```

switch ( sex )
{
    CASE 'Male'      : ...           // processing for male persons
                      break;        // leave switch block
    CASE 'Female'    : ...           // processing for females
                      break;        // leave switch block
    DEFAULT          : error(99);    // indicate error
}

```

`leave` |            The `leave` operand causes the function to leave the `PROCESSING` block. When a `FINAL` block has been defined, the function continues with the `FINAL` block. Otherwise the function terminates and returns the value of the last statement executed.

```

BOOL Person::SendMessage
{
    VARIABLES
        BOOL      to_late = Time > (Time)'17:00';
        BOOL      message_sent = false;
    PROCESSING
        if ( to_late )                               leave;
        ... // prepare and send message
        message_sent = true;
    FINAL
        return(message_sent);
}

```

`return` |            The `return` operand causes to leave the function immediately. When passing an operand to `return`, the value of the passed operand is returned. Otherwise, the value provided by the last operation (statement) executed will be returned. The returned value will be converted to the data type defined for the return value of the function. When no return value type has been defined for the function (`VOID`), no value is returned.

`error` |            The `error` operation causes the function to terminate. When an error block has been defined the function goes to the `ON_ERROR` block. When no `ON_ERROR` block has been defined, the function continues with the `FINAL` block, when being defined. When no `FINAL` block has been defined, the execution of the function terminates and the value of the last statement executed is returned to the calling function.

```

BOOL Person::SendMessage
{
VARIABLES
    BOOL    to_late = Time > (Time)'17:00';
PROCESSING
    if ( to_late )                error(99);
    ...                          // prepare and send message
    message_sent = true;
FINAL
    return(true);
}

```

You may pass an error code to the `error` operation, which can be retrieved from the global variable `_LastErrorCode`.

`exception | EXCEPTION` The `exception` operation causes the function to terminate and to continue with the next `ON_ERROR` block available in the calling hierarchy. When an error block has been defined the function continues with the `ON_ERROR` block. Otherwise it continues with on the `ON_ERROR` block of the next higher function in the calling hierarchy.

```

BOOL Person::SendMessage ( STRING receiver )
{
    if ( !Persons(receiver).exist )    exception(99);
    ...                                // prepare and send message
FINAL
    return(true);
}

```

You may pass an exception code to the `exception` operation, which can be retrieved from the global variable `_LastException`.



## 8.1.2 Built-in Operations

Built-in operations are provided for performing basic operations. OSI supports conversion operations, default logical, text and arithmetical operations, assignment operations and standard query operations.

### Conversion operations

OSI provides implicit conversion whenever required. Implicit conversion is required always, when an operation is performed with operands, the type of which does not match.

```
Message( "Anton " + 5 ); // prints: Anton 5
```

E.g. when adding a number to a string, one operand must be converted. OSI always converts the right operand. Thus, the following function:

```
Message( 5 + "Anton " ); // prints: 5
```

Results in a number (5), since converting the string "Anton " to a number returns 0.

For solving problems, where implicit conversion is not sufficient, cast operators can be used.

```
Message( (STRING)number + "Anton " ); // prints: 5Anton
```

OSI supports conversion between most data types. Excluded from conversion are

VOID, BLOB, BIT

Conversion is supported between basic data types and user-defined data types (enumerated data type – EDT, complex data type – CDT):

CHAR,	→	MEMO, BOOL, INT, REAL,
STRING		DATE, TIME, DATETIME, GUID, EDT, CDT <sup>1</sup>
MEMO	→	STRING, CHAR, BOOL
INT	→	STRING, CHAR, BOOL, TIME, DATE, REAL, EDT
REAL	→	STRING, CHAR, BOOL, INT
DATE, TIME	→	STRING, CHAR, BOOL, INT
DATETIME	→	STRING, CHAR, BOOL, DATE, TIME, CDT
BOOL	→	STRING, CHAR, INT, REAL
GUID	→	STRING
EDT	→	STRING, CHAR, BOOL, INT, REAL
CDT	→	STRING, CHAR, DATETIME, CDT <sup>2</sup>

Converting complex data type to complex data types is performed by names, i.e. all properties are converted, which have the same name in source and target. When properties have complex data types, they are, again, converted by names of the properties in the corresponding complex data type.

Collection properties are converted by copying and converting all collection from the source data type to the target data type.

## Logical operations

Logical or Boolean operations are operations, which return TRUE or FALSE always. OSI supports the following logical operations:

---

<sup>1</sup> Converting string to complex data type requires an extended SDF (ESDF) string type.

<sup>2</sup> Complex data type is converted to extended SDF string format

<, >, >=,  
<=, ==, !=

Comparison operations are used to compare two values, where an order of values is implied depending on the data type. OSI provides implicit data conversion, when data types are not comparable by converting the right operand into the data type of the left operand.

NOT (!)

The NOT operator is supported for all data types that allow conversion to `BOOL`. Data is converted to `BOOL` before applying the operator.

AND (&&) ,  
XOR (^) ,  
OR (||) ,

As well as the NOT operator, all operands are converted into `BOOL` before applying the operation to the operands. Operands, which cannot be converted into `BOOL`, are considered having the value `FALSE`

### Arithmetical operations

Arithmetical operations are operations, which return a numerical value always. OSI supports the following arithmetical operations:

-

As unary operator, the minus operator multiplies the right operand with -1. When the right operand is not numeric, OSI tries to convert it into a `REAL` number. When this is not possible, 0 is assumed as value.

+, -, \*, /,  
%, ^

Binary arithmetical operations require a left operand with a numeric data type (`INT`, `REAL`). The result is provided in the data format of the left operand. Before, OSI tries to convert the right operand into the data format of the left operand. When this is not possible, 0 is assumed as right operand.

### String operations

String operations are supported by using arithmetical operators. Beside those simple string operations additional string functions are provided, which are described under "Built-in Functions".

+

The +-operator allows concatenating two strings. It requires a text field (`STRING`, `CHAR`, `MEMO`) as left-side operand. The right side operand is converted into a `STRING` data type, when this is not yet the case.

-

The operator removes all occurrences of operand 2 in operand. The operation requires a text operand as left operand and converts the right side operand into `STRING`, if required.

```
Message("Paul Miller" - "l" ); // prints: Pau Mier
```

## Date and Time operations

+ , -

When the first operand for a + or – operation is a `Date` or `Time` value, a date or time operation is performed.

You may add or subtract an integer value to/from a date or time value. When the right operand is not an integer value, it is converted into an integer. If this is not possible, the right operand is assumed to be 0.

Adding or subtracting an integer value to/from date means increasing/reducing the date by the given number of days. You may also subtract date values to get the difference between two dates. In this case, the result is not really a date but an integer.

```
Message( Date - (DATE)"2005/10/01" ); // prints: 36
Message( Date - 1 ); // prints: 2005/11/05
```

Adding or subtracting an integer value to/from time means increasing/reducing the left time value by 1/100 seconds \* right side operand. This works properly also, when the right side operand is a time value, i.e. you may add or subtract time values.

## Assignment operations

=

OSI supports two types of assignment operations: assignment by value and assignment by reference. In most cases, the proper operation results from the value definition, depending on whether the operands are value or reference operands.

Value assignment will assign the value of the right operand to the left operand. This is usually done by copying the value.

```
STRING      name;
PROCESSING
...
name = "Miller";
```

When assigning values, data conversion is performed whenever required. Data is always converted to the type of the left operand.

When assigning collections by value, instances are copied, when the left operand is the owner of the instances (owning collection). Otherwise, references to instances are copied. Thus, as result one may get two different collections referring to the same set of instances.

The left hand operand for a value assignment can be a value or reference variable.

&=

Assignment by reference will not copy the value to the left operand, but a reference to the value, only. Thus, right and left operand will refer to the same value and changes in the value become visible in both operands.

Reference assignments do not support data conversion, but require data compatibility between left and right operand. Data can be assigned by reference, only, when the left operand has been declared as reference variable and one of the following conditions is true:

- the left operand is of type ANY (VOID)
- the right operand's type inherits from the left operands type
- left and right operand have the same type

```
BOOL      Test ( Person &pInst )
{
VARIABLES
    STRING          &fname;
PROCESSING
    ...
    fname &= pInst.name;    // reference assignment
    fname = "Miller";      // value assignment, updates pInst.name
    ...
}
```

The reference assignment in the example assigns the name of the Person instance to the local fname variable. The following value assignment assigns "Miller" to the local fname variable, but also to pInst.name, which is referenced by fname.

Parameters passed by reference are considered as reference variables and you may assign a different reference to the variable. In this case, however, the value passed by reference to the function will remain.

When assigning an access path by reference to a variable, the result of the operation is assigned to the variable. This makes a big difference, when using an access path e.g. in a loop.

```
SET<Person>          &children;
PROCESSING
    children &= Persons.children();    // children collection
    while ( children.next )           // iterates through result
    ...
    while ( Person().children().next ) // iterates through path
    ...
}
```

When assigning the access path result to the reference variable, it will be calculated only once. When using an access path directly in a loop, it will be calculated only, when the calling object for the path changes.

## Set operations

Applying arithmetical or Boolean operators on collections allows performing algebraic set operations. Set operations will create new collections, which refer to a selected number of already existing instances, i.e. set operations do not create new instances.

All operations can be called as built-in functions as well and are described there.

+ , |

The **union** operation combines two collections, i.e. it creates a new collection, which contains the instances from both operand collections. The type of the result depends on the collection type of the operands:

- Type of the first operand, when types of both operands are identical
- ANY (VOID), when operand types are not compatible
- Type of first operand, when the second operand inherits the type from the first operand.

The result collection becomes weak-typed, when the operand types are not identical or when at least one of the operands is weak-typed.

The operation is using the distinct option, when the first collection is a ordered collection with a unique key. In this case, the result collection is an ordered collection with unique keys as well.

-

The **minus** operation creates a copy from the first operand and removes all instances that exist in the second operand. The type and attributes (weak-typed, ordered, and unique) of the result collection are the same as type and attributes of the first operand.

When the first operand is ordered by unique key, the operation is performed by key, otherwise by object identity.

&

The **intersect** operation provides all instances, which appear in the first and in the second operand collection. The type and attributes (weak-typed, ordered, and unique) of the result collection are the same as type and attributes of the first operand.

When the first operand is ordered by unique key, the operation is performed by key, otherwise by object identity.

## Aggregation functions

Besides user-defined aggregation functions, ODABA supports a number of built-in aggregation functions:

- minimum
- maximum
- sum
- average
- variance
- deviation (standard deviation)
- statistic (provides count, sum and square sum)

Usually, aggregation functions need a preceding collection name referring to the collection of instances to be aggregated. In order to calculate the aggregated value the function iterates through the collection (calling object). The parameter for the aggregation function is the attribute to be aggregated.

statistic

The statistic function is required mainly for providing statistics in an aggregation model. The function returns a statistic object (**SDB\_Statistic**), which contains an integer attribute `x0` for count, a float point attribute `x1` for sum and another float pint attribute `x2` for square sum. Moreover, the statistic object provides an OSI interface for obtaining calculated statistical values for average, variance and deviation.

The function may apply on numeric values or on statistic object instances (**SDB\_Statistic**). When applying the function on statistic object instances, `x0`, `x1` and `x2` from both instances are added. This is typical the case on higher aggregation levels, which allows applying all supported statistic functions on any aggregation level.

## Optimizing

When calling different aggregation functions for the same collection, each aggregation function will read all collection instances, which may become quite inefficient. In order to optimize aggregation in an OSI function, one may define a variable or parameter with the name `partition`. When not preceding the aggregation function with a collection name, `partition` is assumed as default calling object and all aggregation functions with default calling object are evaluated by one collection iteration.

## Views

When calling aggregation function in views that contain a `GROUP BY` operation, a `partition` collection is created implicitly containing grouped instances. In this case, aggregation functions are optimized also, when not being preceded by a calling object. Optimizing aggregation is not done, when the aggregation function call is part of a more complex expression as

```
avr = sum(income)/partition.count()
```

In order to solve this problem, one may assign the aggregation function result to a view attribute and defining the operation result as transient attribute

```
SELECT ( sex, sum_inc = sum(income), min_inc = minimum(income),  
        transient int(10,2) avr = sum_inc/partition.count() )  
FROM ( Persons )  
GROUP BY ( sex );
```



### 8.1.3 Conditional operands

Conditional operands allow evaluating a value depending on a defined set of conditions (cases). The operand following the first condition that returns true will be returned.

```
age_group = ( age < 16 ? 'teenager'   :  
              age < 30 ? 'twen'     :  
              age < 50 ? 'middle-aged':  
              'oldie'               );
```

Conditional operands may be used in assignments, or passed as parameters. One may also define conditional operands when initializing variables.

Wherever conditional operands are used, they require parenthesis at beginning and end.

## 8.1.4 Query Operations

Query operations are operations, which are supported in a traditional database `SELECT` query.

```
SELECT ( parm_list ) FROM ( parm_list )
      [ WHERE ( condition ) ]
      [ GROUP BY ( parm_list ) ]
      [ HAVING ( condition ) ]
      [ ORDER BY ( parm_list ) ]
      [ TO FILE | TO DATABASE ( out_spec ) ] ;
```

The traditional query format is supported for convenience, only. In contrast to traditional query languages, OSI considers each part of the operation as independent operation or function, which can be called for any collection. Within OSI, one would rather define an operation path, which expresses the same query as sequence of operations, which would look like:

```
[FROM(parm_list)] [.WHERE(condition)] [.GROUP(parm_list)]
[.HAVING(condition)].SELECT(parm_list)
[.ORDER(parm_list)];
```

In case of simple `FROM` source (single collection property), one could also replace the `FROM` operation by the collection name (calling object).

OSI allows combining query operations in any order. The only condition is, that query operations are preceded by a collection operand, i.e. you could write e.g.:

```
Persons.GROUP( string inc_group = (income < 1000 ? 'poor' :
                                   income < 5000 ? 'medium':
                                   income < 100000 ? 'rich':
                                   'very rich') );
```

Query operations can be defined as any other operand in an access path, but one may also refer to traditional query statements as mentioned at the beginning of this chapter.

### **FROM | from**

The `FROM` operation allows combining a number of collections in a product set (outer join). Also, the `FROM` operation supports inner join operations. Usually, it does not make sense, calling the `FROM` operation with a single collection, which will be returned unchanged from the operation.

Typically, the `FROM` operation refers to multiple sources as in the following example:

```
FROM ( Persons, Companies ) ...
```

In this case, the source is constructed from the product set of the collections passed to the operation. The result can be considered as complex data type inheriting from the data types of the referenced collections, `Person` and `Company` in this case, i.e. subsequent operations (e.g. conditions in a subsequent `WHERE` clause) may refer to members in `Person/Company` instances.

For accessing ambiguous property names in the path, the property references can be preceded by the collection name of the referenced collection, e.g. `Persons.name` or `Companies.name`. You may, however, also assign an explicit collection name to each parameter for the operation.

```
FROM ( p = Persons, c = Companies ) ...
```

In this case, property references can be expressed as `p.name` or `c.name`. As long as property names are unique, the collection qualifier can be omitted, since the `FROM` data type inherits from data types from operands.

Instead of a complex data type, you may define a property path.

```
FROM ( Persons.company ) ...
```

Here, the view source again inherits from `Person` and `Company` (the type of `company` in `Persons.company`). From a structural point of view, this would be the same as in the definition above, but the property path implies already a selection of `Person/Company` couples, where the person is employed in the company, i.e. it corresponds to an inner join operation. Thus, a property path works much faster than selecting from a product set.

For accessing ambiguous property names in this example, the property references can be preceded by the property name of the referenced collection, e.g. `companies.name`.

The `FROM` operation may also refer to access paths or expressions.

```
FROM ( p = Person.GetRelatives(),  
      c = {Companies.Large() + Companies.Medium();} ) ...
```

When referring to an access path or inline function, the included collections should get a name in the `FROM` operation. Otherwise it becomes difficult to access ambiguous names in the source. Inline functions must always be enclosed in { ... }

The result consists of the product set of the collections returned as result from the access path or expression.

**WHERE | where**  
**HAVING| having**

For selecting elements from a collection, a filter condition can be defined using the `WHERE` operation.

```
FROM ( Persons ) WHERE ( company.count() == 0 &&
                             age_years >= 65 ) ...
```

In an operation path, the `WHERE` operation always applies on the preceding collection. Thus, the distinction between `WHERE` and `HAVING` becomes obsolete. Nevertheless, you may use `WHERE` as well as `HAVING`. Both operate in the same way.

In a `SELECT` statement, however, the `HAVING` operation applies on the `SELECT` data type while the `WHERE` operation applies on the `FROM` data type.

The `WHERE` operation will not alter the data type of the instances, i.e. the output instances for the operation have the same structure as the input. This is also valid for the collection attributes (e.g. weak-typed or sort order).

**SELECT |**  
**select**

The `SELECT` operation allows defining an implicit defined complex data type. The `SELECT` allows defining any number of attributes or collection properties by defining the data type, the property name and an assignment expression defining the value for the property.

```
... . SELECT ( parameter [{\,' parameter}(*)] );
```

Property definitions (`parameter`) in a select statement passed as parameters are similar to view member definitions in the view definition, except that they are separated by comma and only support attributes but no reference properties.

As operands one may define simple property names defined in the view source, but also operands or member definitions:

```
SELECT ( sex, // attribute reference
         string age = age_group, // attribute definition
         int cnt = sum(1) // named operation
         int(10,2) avr_inc = average(income) ) // operation source
```

One may omit data type declarations for `SELECT` parameters. In this case, the data type is determined by the source data type. The source is a property name, an access path or an expression valid for the data type of the calling object for the `SELECT` operation.

The simplest example for a `SELECT` operation is:

```
SELECT ( * )
```

Which does nothing else than returning the instances as being defined in the source operand (calling object or `FROM` operation) for the `SELECT` operation. This operation does not make sense in an access path but is supported because it is required sometimes in traditional `SELECT` statement.

Assignment operand

Each property definition in the select statement may have a name and an assignment operand. The name is optional and can be omitted, when the assignment operand is a simple property in the source for the select.

```
Person.select( name, first_name );  
Person.select( n = name, fn = first_name );
```

The example above refers to two simple operands defined in the source and the name for the output is taken from the operand in this case, i.e. both definitions are identical. When operands become more complicate, properties should be named.

Data conversion

One may request explicit data conversion by defining a data type for the property name.

```
Person.select( STRING(20) name = name, ... );
```

In this case, data conversion is performed according to the common data conversion rules.

Data source

The source for a `SELECT` operation (view source) in an access path is the result of the preceding operand. In a traditional `SELECT` statement, the view source is the result of the `FROM` operation when no grouping has been defined or the result of the `GROUP BY` operation otherwise.

Aggregation

The `SELECT` statement supports aggregation functions. When the `SELECT` statement contains a `GROUP BY` operation, aggregation function may refer to default calling object `partition`. When not grouping instances, one may assign a collection property to view member in order to use optimizing feature for default aggregations.

```
SELECT ( sex, name, first_name
        set<Person> &partition = children,
        int(10,2) dev_inc = deviation(income),
        int(10,2) avr_inc = average(income) )
```

When no `partition` property has been defined, aggregation functions need a preceding collection name referring to the collection of instances to be aggregated. The parameter to be passed to the aggregation function is a member of the `FROM` or calling object instance. The parameter passed to the aggregation function is either an attribute name defined in the data type of the aggregation collection or a valid expression for the aggregation collection data type.

```
Companies.SELECT (
    INT(10,2) sum_income = employees.sum(income),
    ...
);
```

This allows providing aggregated values for different collections.

`partition`

In order to provide more efficient aggregations, one may explicitly define a default aggregation collection `partition`. Aggregating data in the `partition` collection causes a single iteration through the `partition` collection regardless on the number of aggregation functions called.

```
Companies.SELECT (
    partition = persons,
    INT(10,2) avr_inc = average(income),
    INT(10,2) dev_inc = deviation(income),
    ...
);
```

When explicitly defining a `partition` property, it has to be defined before referring to the first aggregation function.

As data source for `partition` one may also define an expression in the context of the `SELECT` parent. In case of `GROUP` operations, a `partition` collection is created implicitly, i.e. all default aggregation function calls refer to the implicitly defined `partition`.

For a `GROUP BY` operation, the `SELECT` operation could be defined as in the following example:

```

Person.GROUP(name).SELECT(
    family_name = name,
    INT(10) sum_income = partition.sum(income),
    SET<Person> adults = partition.WHERE(age > 18),
);

```

The expression above aggregates the income for each family, which has been grouped by the preceding `GROUP` operation. Each instance in the result collection gets the family name, which is the grouping value. Finally, a collection of adult persons is created for each family name and added as collection to the result instance.

Since `partition` is a default property for aggregation operations, it can be omitted in connection with aggregation functions, i.e. the statement above could be written as::

```

...
    INT(10) sum_income = sum(income),
    SET<Person> adults = partition.WHERE(age > 10),
);

```

Since `WHERE` is not an aggregation function, `partition` cannot be omitted for the `adults` assignment operand. Supported aggregation functions are described in chapter “Property handle functions”

**ORDER [BY] |**  
**order [by]**

Ordering allows changing the sort order of a collection either to a key name defined for the complex data type of collection instances or to a list of attributes passed as operands.

When a persistent index has been defined for the key name passed as operand, the access key for the collection will be changed to the defined index. Otherwise, a temporary key will be defined and a temporary sort order will be created for the collection. Then, the access key will be set to the temporary index.

When referring to the operation from within an operation path, the single word operand `GROUP` (or `group`) has to be used.

```

Persons.ORDER( pk ) ... // orders collection by key pk
Select(*) ORDER BY( name, first_name ) ... // orders by names

```

**GROUP [BY] |**  
**group [by]**

Grouping provides a way of aggregating data. In an access path, the instances in the preceding operand collection are grouped. In a traditional `SELECT` statement, instances defined in `by FROM ... SELECT ... WHERE` operations are grouped.

When referring to the operation from within an operation path, the single word operand `GROUP` (or `group`) has to be used.

`partition`

For each grouping instance, data source instances belonging to the group are collected in the `partition` collection. The `partition` property will be implicitly defined for a grouping operation. The type of instances in the `partition` collection corresponds to the complex data type resulting from the calling object or `FROM` operation.

For each grouping instance, the `partition` collection collects all instances from the source operand that return the same grouping key for the grouping attributes.

When the source data type defines an identifying key, the `partition` collection will be ordered according to the identifying key of the source data type. When the source data collection is week-typed, `partition` becomes week-typed as well. `partition` will be the instance owner, when instances passed to the operation are not persistent instances. Otherwise, `partition` will reference instances, only.

The members defined in the grouping parameter list (parameters for the `GROUP BY` operation) have to be attributes of the source operand data type. The attribute type can be a basic data type, but also a user-defined data type (enumerated or complex data type). In order to group by ad-hoc classifications, an appropriate transient attribute evaluating the value for the ad-hoc classification has to be defined in the data type of the source operand (e.g. in the `SELECT` operation).

```
Persons.GROUP( address, sex) ...  
Select (*) FROM (Persons) GROUP BY (address, sex) ...
```

The grouping operation creates an instance in the result set, which consists of grouping attributes defining a unique key and the `partition` collection.

In the example above, one instance will be created for each city where the person lives, grouping all persons living in the same city in the `partition` reference collection.

Grouping attributes automatically become the identifying key and the only order key for the result collection.



**TO FILE |**  
**to file**  
**TO DATABASE |**  
**to database**

Usually, the result of a query or operation is assigned to a variable in the function. In some cases, one wants, however, store the result in an external file or in another database. For exporting the result, two operations are provided, which can apply on collection operands in an access path, but which can also be used as OSI extension in a traditional `SELECT` statement.

When referring to the operation from within an operation path, the single word operand **toFile** has to be used.

File output

The output to file operation will export the collection defined in the source operand or in the `SELECT` statement to a file or to console.

```
Persons()->toFile ( Path='Console', FieldSeparator=',',  
                  Definition='extPerson', Source='store',  
                  FieldSeparator=';', HeadLine=true ) ;
```

The output to file operation requires an operand list after the `TO_FILE` keyword. The operand list requires parameters with specific parameter names, which allows omitting parameters, which are set to the default value. Moreover, the sequence of parameters does not matter.

`Path='complete_path_name'` - The path name points to the location for storing the file. The complete path name should be enclosed in quotes (single or double) to avoid misinterpretations. The default for path is 'Console', which will direct the output to the screen.

`PathOption='option_name'` - Alternatively to the file path an option name may be defined. The option must be set to the file path. The option name is also used as extent name for the external file collection.

`FieldSeparator=';'` - This is the field separator, when another than ',' should be used. The field separator can be passed as string but also as hexa-decimal value (e.g. 9 for tab), which corresponds to the character values.

`FileType='csv'` - File storage format (**xml**, **csv**, **oif**, **esdf**) when this is different from the file name extension.



The input file operation requires an operand list after the `TO FILE` keyword. The operand list requires parameters with specific parameter names, which allows omitting parameters, which are set to the default value. Also, the sequence of parameters does not matter.

`Path='complete_path_name'` - The path name points to the location for loading the file. The complete path name should be enclosed in quotes (single or double) to avoid misinterpretations. The default for path is 'Console', which will direct the output to the screen.

`PathOption='option_name'` - Alternatively to the file path an option name may be defined. The option must be set to the file path. The option name is also used as extent name for the external file collection.

`FieldSeparator=';'` - This is the field separator, when another than ',' should be used. The field separator can be passed as string but also as hexa-decimal value (e.g. 9 for tab), which corresponds to the character values.

`FileType='csv'` - File storage format (**xml**, **csv**, **oif**, **esdf**) when this is different from the file name extension.

`Definition='type_name_path'` - Name of data type definition defining for the collection in database. Instead of a type definition name, a file name for a definition file (ODL, ESDF or XML format) may be defined.

`Source='source_name'` - Property definitions in type definitions for external data sources may contain any number of source definitions. Source definition to be used for import may be identified by source name. When no source name has been passed, the first source definition for each property is used. File type definitions contain only one source for each property, which is not named, i.e. source name must not be defined when providing an external (file) definition.

## 8.2 Using transient variables

In order to store run-time information or derived data in persistent class instances, transient attributes or references might be defined. When defining transient instances as e.g. context class instances, references cannot be defined. In this case, all attributes are considered by default as transient.

Transient attributes are filled automatically, when a source has been defined (OSI expression) and when the attribute is accessed. Transient attributes maintained by the application, have to be set explicitly. In order to refer to collections or persistent instances, SET variables have to be defined.

```
SET<Person>    persen_set;  
SET<Person>    &persen_ref;
```

There is no big difference in defining direct or reference variables for collections, except, that one may assign a collection to reference variables, only.

When defining transient variables, those are not valid (isValid() returns false) as long as they are opened or a valid collection has been assigned.

```
persen_set.open(database, 'Person', AccessModes::Read);  
persen_set.use(Person); // use person extent  
person_ref &= Person; // reference variables, only
```

Opening or assigning a collection to a transient variable reserves the property handle for the variable. This becomes necessary when subordinated properties are returned to the caller. When not reserving top property handles, i.e. when defining those as local variables, they will be destroyed when leaving the function or moved to the garbage collector and subordinated properties may become invalid.

## 8.3 Operation paths

An operation path is an access path, which includes one or more operations. Operations are built-in functions as `SELECT` or `INTERSECT`, but also user-defined functions or context functions.

Operation paths can be defined as operation path variables or as operand in a function.

Operation path includes access paths and property paths, which refer to property names, only, but not to operation names.

### Operation path elements

Operation paths consist of one or more elements separated by path separators (see “Language Reference - access path”). Elements in an operation path are

- Property references
- Operation calls
- Collection elements

### Property reference

A property reference is a valid property name in the path referring to a property in the scope of the current function or to an Extent (first path element) or to a valid property name in the scope of the preceding operand (second and following elements).

```
Person.children
```

Property references can be defined as

- Simple property
- Iteration property
- Location property

Simple property references as in the example above are used in their current state, i.e. no position operation is performed when executing or initializing the path.

Thus, in the example above the path refers to the collection of children for the person selected in the preceding person collection.

Iteration properties are defined by appending `()` to the property name.

```
Person()
```

An iteration property in an operation path caused the path to iterate over the instances in the collection referred to by the property name.

Location properties are defined by appending an instance identification as number (position in the collection) or as sort key value:

```
Person('0077').children(0) // first child of person 0077
```

Defining a location property will cause the path to locate the instance addressed by the instance identification.

Operation call

An operation call is a path element, which refers to

- An OSI function
- A context function
- A built-in function

```
Person.count
```

Operation calls refer to operation names which are valid in the context of the current function (first element) or in the context of the preceding path element.

Operations are defined as instance or set operations. Instance operations operate on each instance passed by the preceding function

```
Person().Print
```

The example above will print out each person from the Person extent.

When following an iteration or location property, the operation must be an instance operation. When following a simple property the operation might be a collection operation operating on the preceding collection or an instance operation operating on the instance selected in the preceding collection.

```
Person.PrintAll
```

The example above will print out all persons in the Person extent. In contrast to the prior example, the function does the iteration over the person collection and not the operation path.

The result of an operation might be a single instance (or elementary value) or a collection. Depending on the return type, the operation behaves as iteration element (collection) or location element (single instance) in the path.

**Operation parameters**

When passing parameters to an operation call, parameter values or expressions must be valid in the scope of the function defining the path (first element) or of the preceding path element. Parameter operands or expressions may refer global and local variables, to constants or to properties defined in scope of the path element.

```
Person().Print(head_line);
```

Usually, parameter variables are looked for in the scope of the calling function and not in the scope of the calling operation element (preceding path element). One may, however, change the parameter scope to the function scope by using the @ element separator instead of the dot.

```
Person()@Print(head_line, company_name);
```

In this case, `company_name` is found in the context of the calling object for the function, i.e. in the function scope.

**Collection element**

An operation path may define parts of the path as collection elements. Usually, an operation applies on the preceding path element.

```
Person().children.count()
```

In the example above, the operation path returns a children count value for each person, i.e. the `count` function is called for the children collection for each person in the person collection.

```
[Person().children()].count()
```

Defining a collection operand `[]` instead in front of the `count` operation will cause the path to create a collection of all persons children before calling `count`, i.e. the path returns the number of all children of all persons in this case.

**Operation path types**

Operation paths may react differently depending on the way they are defined. Three different operation path types can be defined:

- Iteration path
- Location path
- Execution path

**Iteration path**

An Iteration path contains at least one iteration property, which is a property followed by `()`:

```
Person().children().children()
```

Operations in an iteration path are always considered as iteration elements, i.e. when an operation returns a collection, the path will iterate over the result set of the operation as well.

**Location path** A location path is an operation or access path which addresses exactly one data instance:

```
Person('0077').children(0) // first child of person 0077
```

A location path may contain properties with a value selection and operation resulting in a single instance. When referring to a reference or relationship in a location path, the property name must be followed by an instance identification which is either a number (position in the collection) or a sort key value. Also singular references must contain the instance identification (usually 0) to indicate, the an instance needs to be located.

**Execution path** Usually, an execution path is an operation path with at least one execution element. The execution element is introduced by the -> element separator (execute operator) instead of the dot separator.

```
Person().children()->Print
```

The execution element defines the right most position for executing the path when being initialized. Thus, is does not make much sense to define two execution elements in a path, since the last execution element will be registered as such.

The part right of the execution element may form an iteration or a location path. The left part of the execution element is called once when initializing the path or when the calling object has changed.

The following example illustrates the difference:

```
while ( Person().children.count )  
... do something ;
```

Will call the operation each time when entering the while loop.

```
while ( Person().children->count )  
... do something ;
```

Will call the operation only once and end in an endless loop in this case.



## Initializing the path

An operation path is initialized always, when processing the code defining the path. Thus, it is often more efficient to define an operation path variable then to call the operation path e.g. in a while loop. In general, one should avoid calling operation paths directly in loops, because this is not only inefficient, but may lead to unexpected results because of the re-initialization in each loop iteration.

There are, however, some typical examples, where operation paths in loops make sense:

```
while ( Person.next )
    Person.Print;
```

Since the `next` function changes the state of the person collection, initializing the path will locate the next instance in the person collection. Since the path is neither an iteration nor a location path, it will initialize only.

`next` and `previous` are typical functions called in loops.

Initializing a operation path includes:

- Locating the instance in a location path
- Positioning an iteration path to top
- Executing the path until the execution element (when being defined)
- Calling operations in the path as long as the preceding path element is positioned or the operation is a collection operation.

## Executing the path

Executing an operation path includes iterating through all elements for an iteration path. Path execution does not make sense for location paths, since there is nothing to be executed.

Execution paths are running in the scope of the preceding path element, always, i.e. one cannot change the scope to the current function.

Usually, the execution element is the last element in an execution path. When an execution path, however, has path elements left of the execution element, those are initialized each time after calling the operation.

A path is executed when being initialized the first time. Later, the path will be executed only, when its calling object instance has been changed. Hence, calling an execution path in a loop usually will not cause problems, since it will be initialized (executed) only once.

Executing a path is automatically done, when defining an execution element in the path. When an execution element has been defined in the path, this will cause iterating over the part left of the execution element.

```
Person()->Print;
```

The example above has the same effect as the while loop in the prior example. Because of the execution element `Print()` and the iteration part left of it the path automatically iterates over persons executing the `Print()` operation for each person instance.

You may also define execution elements for a location path. This allows distinguish between class members and methods.

```
Person->first;  
Person.first;
```

Both examples call `first` in order to locate the first person instance. When, however, an attribute `first` has been defined for the `Person` class, only the line will call the property handle function `first`, while the second line refers to the class member.

### Operation path variables

When assigning a query as initial value to a variable, the query is defined but not executed immediately.

```
SET<Person> grand_children =  
Persons().children().children().where(age < 18);
```

In the example above, the view is assigned to the variable `grand_children` and initialized. Later on, in the processing section of the function you may refer to the variable and iterate through the collection defined by the variable.

Initializing an operation path includes locating the referenced instance, when the path is referring to a specific instance, i.e. when the path is not an iteration path.

```
while ( grand_children.next )  
grand_children.Print;
```

The operation path `grand_children.next` is initialized each time when calling the `while`

## 8.4 Dynamic function calls

Dynamic function calls is a feature, which allows deciding at runtime, which function has to be invoked. In order to call a function dynamically, the function name has to be passed in a parameter, variable or option variable.

### Local variables and parameter

Calling a function dynamically via local parameter or string variable, the value for the function name string has to be set explicitly before calling the function.

```
STRING      fname;
...
fname = GetFunctionName(); // returns a function name string
CALL fname(param1, ..., paramn);
```

CALL only accepts the variable name followed by the function parameter list. The parameter list must match the function patterns of the function to be called. Otherwise, parameter errors may occur.

### Option variables

In order to control execution via option variables, functions may also be called by option variables set in the application or configuration file..

```
CALL %FNAME%(param1, ..., paramn);
```

When the function name is not valid or when the parameter list does not match the function definition, CALL will terminate with error.

## 8.5 Built-in Class Functions

Built-in class functions are functions provided for different access classes. Most functions are provided for access classes.

### Restrictions

This does not include operators, which are handled directly by OSI. Also, constructors are not supported, except the dummy constructors with no parameters.

Instead of calling a constructor, one may call the appropriate open functions, which are provided for all access classes.

### Interface functions

OSI supports most of the functions documented in the ODABA interface classes. Besides access classes, this includes metadata functions, but also options or date and time functions.

### Access classes

In order to provide access to the database, access class functions are supported.

- **Application**
- **Dictionary**
- **Database**
- **ObjectSpace**
- **Property**
- **Value**

Each variable in an OSI function automatically supports Property and

### Metadata

Metadata object types cannot be constructed at all, but can be obtained from access objects.

- **TypeDefinition**
- **PropertyDefinition**
- **IndexDefinition**
- **EnumeratorDefinition**

### Option

Although, options can be accessed directly from within OSI functions. The `option` class has got an interface, which provides more specific functionality for accessing options. In order to use option functions, options have to be created as empty objects and opened afterwards (open function).

### Date/Time

Date/Time objects can be created but have to be initialized by assignment.

- **Date**
- **Time**
- **DateTime**

## Service classes

Additional classes have been provided in order to support specific services as file access.

- **File** - File access
- **BinaryFile** - Binary file access
- **TextFile** - Text file access
- **IniFile** - Configuration or ini-file
- **ZipArchive** - ZIP File Archive
- **MP3File** - MP3File handle
- **MP3Header** - MP3 header handle
- **MP3Frame** - MP3Frame handle
- **XMLString** - XML string
- **HTTP** - Internet protocol
- **Email** - Sending and receiving emails
- **BNFParser** - Parser for files or strings with generic BNF syntax
- **BNFNode** - BNF tree node

## SystemClass

Common features are supported by the **SystemClass**, which provides some functions in order to support handling template result, date/time and other features. The system class is provided as built-in class, i.e. all functions can be referenced without being prefixed by the scope operator.

When function names become ambiguous, the scope operator has to be used:

```
Message("Hello world");
SystemClass::Message("Hello world");
```

**SystemClass** functions are static functions and do not need a calling object instance. .

## 9 OSI Templates

OSI templates are a specific way of defining a function. Usually, OSI templates are used, when having a sort of text template, which is to be filled with data from the database.

Since one can use the `WriteResult` or `FileHandle::Out` functions in a OSI function, template functions are not really required, but are useful in many cases. In contrast to OSI functions, template functions are easier to read and easier to write.

OSI templates can be considered as inverse functions, but there are some restrictions compared with functions.

```
$template string Test()$  
Dear $if (sex == 'male')$Mr. $else$Mrs. $end$ $family_name$,  
We just got your question concerning $product$, which you are  
using since $buing_date$. We have forwarded your problem to  
$responsible(0).first_name$\ $responsible(0).second_name$.  
You will get an response during the next three days.  
...  
$return TemplateString$  
$end$
```

Expressing the template above as a OSI function would look like:

```
function string Test () {  
    WriteResult("Dear ",false);  
    if (sex == 'male') WriteResult("Mr. ",false);  
    else WriteResult("Mrs. ",false);  
    WriteResult(family_name,false);  
    WriteResult(", ",false);  
    WriteResult("We just got your question concerning ",false);  
    WriteResult(product,false);  
    WriteResult("which you are using since ",false);  
    WriteResult(buing_date,false);  
    WriteResult(".",false);  
    WriteResult("We have forwarded your problem to ",false);  
    WriteResult(responsible(0).first_name,false);  
    WriteResult(" ",false);  
    WriteResult(responsible(0).second_name,false);  
    WriteResult("You will get an response during the next three  
days. \n...\n",false);  
  
    return TemplateString;  
}
```

You may call templates from within a template etc. The

template result will be generated as templates are called. Since you may access the template result at any time, you may also update the generated result during the generation process.

Since templates are just a simplified way of defining text functions, you may also mix templates and OSI functions or create a template result just by calling OSI functions.

## 9.1 ASCII templates

OSI templates can be considered as inverse OSI functions, but there are some restrictions compared with OSI functions.

### Special characters

One may create any sort of text output using template functions. The only reserved characters in a template are \$ and \. When you have \$ or \ in the text, you need to add a \ before, i.e. \\$ or \\ in the text will be converted to \$ or \ respectively.

### Fill characters

Blanks, new lines (0x0D0A or 0x0A) and tabs (0x09) are considered as fill characters.

In some cases, fill characters are not displayed properly. Usually, all fill characters between text and embedded expressions are considered as part of the text constant.

```
...
$responsible(0).first_name$ $responsible(0).second_name$
...
```

The text above will generate the following statements

```
WriteResult(responsible(0).first_name,false);
WriteResult(" ",false);
WriteResult(responsible(0).second_name,false);
```

This will also include a blank between first and family name.

Line breaks or tabs between embedded expressions or at the beginning of the text are also considered as fill characters to be displayed in the result.

```
...
$responsible(0).first_name$
$responsible(0).second_name$
...
```

This will result in:

```
WriteResult(responsible(0).first_name,false);
WriteResult("\n",false);
WriteResult(responsible(0).second_name,false);
```

For ignoring line breaks between embedded expressions you may use the line connector \. Adding a \ at the end of a line causes the template to consider the next line as continuation of the current line. This also allows inserting line breaks in the template text, which may increase the readability of the template.



```
...
This is a longer text constant to be displayed in the result \
in a single line. To make the template more readable, we can \
add '\\\'' in the template text before line break.
...
```

#### Fill character sequences

All new lines found in the template before and after text constants are considered as text constants and included in the result. Thus, the following template fragment

```
You will get an response during the next three days.
...
```

Will generate function code as follows:

```
WriteResult("You will get an response during the next three
days. \n...\n", false);
```

One may also add explicitly defined fill characters as \n, \t or \, which will have the same effect.

```
You will get an response during the next three days.\n...\n
```

Will generate the same function code as above.

```
WriteResult(You will get an response during the next three
days. \n...\n", false);
```

Defining reserved sequences as part of the text can be done by inserting an additional \ in front of the sequence:

```
You will get an response during the next three days.\\n...\\n
```

Will generate the same function code as above.

```
WriteResult(You will get an response during the next three
days. \\n...\\n", false);
```

Later on, Write result will convert double backslash \\ into single once and the final output appears as:

```
You will get an response during the next three days.\n...\n
```

#### Comments

Comments are line comments introduced by //. Comments can be placed at the end of a line, only, i.e. any text after the comment-begin is considered as part of the comment until the line end.

Comments within a text constant are not recognized as such but considered as part of the text constant. Adding comments in a template is possible at the beginning of the template or after embedded code or immediately after a fill character sequence.

```
// this is a valid line comment (at beginning of template)
This is part of the template text // and this also
$Data$ // display current data - valid comment
```

In order to append a comment at the end of a text

constant, you have to insert an explicit line break before the comment

```
This is part of the template text \n// now it's a comment  
$Data$ // display current data - valid comment
```

For generating // sequences as template text, you may use V, which will be converted to //.

```
This is part of the template text \/ generated to the text as //  
$Data$ // display current data - valid comment
```

## 9.2 HTML Templates

HTML templates In case of HTML templates, however, text included in the function requires special treatment, because characters as `<` or `>` need to be converted.

This is automatically done, when defining an HTML template as:

```
<template>
  <header> string Test() </header>
  <processing>
<body>Dear $if (sex == 'male')$Mr. $else$Mrs. $end$ $family_name$,
We just got your question concerning $product$, which you are using
since $buing_date$. We have forwarded you problem to
$responsible(0).first_name$ $responsible(0).second_name$.
You will get an response during the next three days. <br/>
... <br/>
</body>
$return TemplateString$
  </processing>
</template>
```

When converting HTML templates to OSI functions, all text read from the database is converted to HTML by replacing reserved characters. In this case, the following code would be generated:

```
function string Test () {
  WriteResult("<body>Dear ",false);
  if (sex == 'male')
    WriteResult("Mr. ",false);
  else
    WriteResult("Mrs. ",false);
  WriteResult(family_name,true);
  WriteResult(", ",false);
  WriteResult("We just got your question concerning ",false);
  WriteResult(product,true);
  WriteResult("which you are using since ",false);
  WriteResult(buing_date,true);
  WriteResult(".",false);
  WriteResult("We have forwarded you problem to ",false");
  WriteResult(responsible(0).first_name,true);
  WriteResult(responsible(0).second_name,true);
  WriteResult("You will get an response during the next three days.
<br/>\n...<br/>\n</body>");

  return TemplateString;
}
```

The difference is in calling the `WriteResult` function, which passes **true** as second parameter to indicate HTML

conversion.

### **Special characters**

The rules for reserved template characters are the same as for ASCII templates, i.e. you must escape all special characters (\$ or n), which are supposed to appear as such, in the fixed text.

Special HTML characters in the fixed text must be defined in an HTML conform way (e.g. &lt; for <). Transformations are done only for the text read from the database.

In order to suppress HTML conversion, one may define an ASCII template instead. In order to suppress HTML conversion partially, one may modify the generated function code or define an explicit function.

### **New lines**

New lines do not have any effect on the generated HTML page, but may make the generated code more readable. The template does not create line breaks <br/> or paragraphs <p> from new lines. Those must be defined explicit in the fixed text as all the other HTML tags.

## 9.3 Template specifications

In most cases, templates do have a PROCESS section, only. Templates may have, however, also a VARIABLE section. ON\_ERROR and FINAL sections are not supported for templates.

**General structure** ASCII templates can be defined as

```
$template string Test()$
$VARIABLES$
    int        count = 0;
$PROCESS$
    ... template text
$END$
```

HTML templates look a little bit different like

```
<template>
  <header> string Test() </header>
  <variables>
    int        count = 0;
  </variables>
  <process>
    ... template text
  </process>
</template>
```

Note, that an html template must not contain the </process> sequence as fixed text in the process section, since this will terminate the process section. This problem can easily be solved by calling a separate ASCII template, which just produces the </process> sequence.

In general, it is suggested to use ASCII templates rather than HTML templates.

**Template body**

Template text (fixed text) can be entered in the template body (process section). Template text contains fixed text and embedded code.

Fixed text

Fixed text is any sequence of characters (including fill characters as blanks or line breaks) except sequences enclosed in \$...\$, which are called embedded code.

There are three types of imbedded code expressions.

Output expressions

Output expressions are operands (usually database property names), which are enclosed between \$...\$

```
$first_name$
$responsible(0).first_name$
$responsible(0).first_name + ' ' + responsible(0).second_name$
```

Thus, you may enter template call for other templates or operations of any complexity in an output expression, but no statements terminated by semicolon.

The content of an output expression is directly written to the target string. In an HTML environment, it is converted to HTML before.

**Embedded code** Embedded code does not directly create output, but is executed as expression code. Embedded code must be enclosed into `#{...}`

```
#{
  while ( messages.next )
    WriteResult(message.text,false); // no HTML conversion
}#
```

Since code may contain `WriteResult` calls, code may also add data to the template result. This is one way to suppress HTML conversion for special texts, which have already HTML format.

Within embedded code you may refer to template variables defined in the variable section, to object variables, parameters and global variables like in an ordinary function.

**Control sequences** Control sequences are special expressions to control the text generation. Control sequences work similarly to the corresponding function constructs.

**return** The return sequence is required, when the template is going to return a value as defined in the template header.

```
#{return operand#
```

The operand for the return value defines the value returned to the caller. No more text is generated after the return has been executed.

**If else end** The if sequence defines a feature for conditional template generation:

```
#{if condition#
  Text generated when condition is true
#else#
  Text generated when condition is false
#end#
```

The else block is optional, but the `#end#` must be defined in any case.

Note that line breaks after the condition become part of the

fixed text and lead to line breaks in the template result.  
Thus, sometimes line breaks must not be inserted:

```
$if (sex == 'male')$Mr. $else$Mrs. $end$
```

**Switch case end**      The switch block is an enhanced feature for conditional processing, since it allows defining any number of processing path.

```
$switch condition$  
$case operand$  
  Text generated when the case operand matches the switch  
$case operand$  
  Text generated when the case operand matches the switch  
$default$  
  Text generated for other cases  
$end$
```

In order to handle other (or default cases), the switch block may contain a default statement.

```
$switch ( hair_color )$  
$case 'blue'$  
  Blue is not an accepted hair color. Use \"other\", instead.  
$case 'yellow'$  
  Yellow is not an accepted hair color. Use \"other\", instead.  
$default$  
  $hair_color$ is a valid value  
$end$
```

**While end**      While and for allow defining loops over arrays or  
**For end**      collections. Similar to the conditional processing, an end  
statement is required in any case.

```
$while condition$  
  Text generated as long as condition is true  
$end$
```

Note, that the defined block must alter the condition. Otherwise, the loop may never end. Typically, `next` is used to iterate through the collection.

```
$while messages.next $  
  Message is: $messages.text$  
$end$
```

It is, however, also possible to insert imbedded code to change the condition:

```
$while ( count < 10 )$  
  Number is: $count$  
  ${++count;}$  
$end$
```

The same way one may define for-loops according to the for-syntax defined for OSI functions.

## 9.4 Template Result

The template result corresponds to the output created by `WriteResult`. The `WriteResult` function appends the text to the result string, i.e. it will collect the output from several templates.

### Global template string

Template strings are thread variables, i.e. they are created separately for each thread. There is, however, only one template string for each thread, which can also be accessed as global variable `__template_result__`.

```
VARIABLES
global    string    __template_result__;
```

It is, however, not suggested to refer to the template result via the global variable name, since the global variable name might be changed. A better way is referring to the template result via the functions describes below.

### SystemClass support

The `SystemClass` provides some functions in order to support handling the template result.

### WriteResult

The `WriteResult` function will append the data passed to the template string. Non-string values are converted to string according to the common conversion rules.

Write result supports converting data passed in the first parameter to HTML compatible data by converting HTML characters:

```
> → &gt;
< → &lt;
& → &amp;
" → &quot;
```

```
WriteResult(data);           // appends content of data to template
WriteResult(data,false);    // same as above
WriteResult(data,true);     // HTML convection befor append data
```

### ResetResult

The `ResetResult` function will clear the template string. The application is responsible to reset the template string at the beginning or when terminating the processing.

### TemplateString

`TemplateString` is a function that returns the template result as string value. Calling `TemplateString`, one may display the template result or write it to file.



```
// write template result to console
Message(TemplateString);
// or write template result to file
FileOpen(file,path);
file.Out(TemplateString());
file.Close();
```

One may also use `TemplateString` to add data directly to the template string or to reset the template string:

```
VARIABLES
    string      &tstring &= TemplateString() // template string
                                                    // reference

PROCESS
    Message(tstring);
    tstring = '';
    Message(tstring);
    tstring += 'new value';
    Message(tstring);
...

```

The application is responsible to clear data in the template string when no longer being used, e.g. by calling `ResetResult`.

## 9.5 Debug templates

Debugging templates may become a little bit complicate, since error will be detect in the function generated from the template. Hence, the line numbers will not fit exactly to the template position. Even templates containing big amount of expression code may cause problems.

Those can be solved partially by viewing the system output, since when detecting an error in the generated code, OSI automatically writes the generated code to the system output.

# 10 Trace function calls

In order to measure time used for function calls, a trace option may be set. The trace option may be set within OSI functions

**#TRACE** top;

or as option (or environment) variable.

**OSI.TRACE**=top

When being set as option variable, all function calls are traced. When being set within an OSI function, this function will be traced, only.

Tracing function calls allows measuring time but also listing function calls sequences.

## Trace options

Several trace options may be set, which may me also combined separated by comma, as

**#TRACE** all, interface; // in OSI function

**OSI.TRACE**=all, interface ; in configuration/ini file

- top Measures total time for function calls that are called as top functions, which is typically the case when calling a function via OSI utility or when calling an event handler in a context class. Usually this option is used to measure time used by context class functions.
- all Measures total time for all function calls. The measured time is the total time, i.e. when calling sub functions, their time is included. The function does not measure time used for ODABA API function calls.
- interface Measures total time for API function calls.
- hierarchy Measures total time for OSI function calls and time for each function call in hierarchical sequence.

## Trace list

The trace results are displayed in semicolon separates text file OSITrace.csv, which will be written to the directory defined via TRACE option. All time measures are provided in milliseconds. Each line in the file contains following fields:

- level Level of function call, which will have a meaningful value for hierarchy, only. For all other options, the value will be empty.

class	Name of class that implemented the function.
function	Function name.
calls	Number of function calls.
time	Elapsed time, which is the total time for all functions, when level is empty or the time for a single call when level has got a value (option is set to hierarchy).
min	Minimum time for all function calls when line displays a total (empty level) and empty otherwise.
max	Maximum time for all function calls when line displays a total (empty level) and empty otherwise.

# 11 OSI-Debugger

OSI provides a command line debugger, which can be activated by setting the OSI environment variable

**OSI.DEBUG=YES**

Or by passing the debug option `-DB` when calling OSI or OShell. When running OSI scripts without debug mode, debug mode may be activated later by pressing `ctrl-c`. Then, the debugger will halt at the next statement to be processed.

When the debug mode is activated, the execution stops at the first line. You may use the 'go' command to continue execution until the first break point or abnormal termination of a statement. When the debugger stops the execution, the current line will be displayed on the console. The debugger provides several list functions, which allow displaying the current state of variables and objects.

When running GUI applications in debug mode (`OSI.DEBUG=YES`), those must be called via **code** (**code.exe**) and following option has to be set in addition:

`CONSOLE_APPLICATION=YES`

in order to redirect application output to console.

## Options

Some debug options might be set in order to improve debugging.

### Stack limit

For detecting infinitive recursive function calls, the **STACK** option may be set, which limits the number of stack frames for calling OSI functions. Usually, a stack limit of 200 should be enough for running most OSI applications:

**OSI.STACK=100**

When no stack limit has been set, the number of stack frames is not limited.

### Run option

In order to start an OSI application without breaking at first statement, the **RUN** option may be set to true:

**OSI.RUN=YES**

When not being set, the debugger breaks at first OSI statement, which also might be a selection condition or a

context function.

## 11.1 Breakpoints

Break points can be set in advance or during debugging. Predefined breakpoints are set by inserting '#' in front of the statement where the execution should break.

```
while ( next )
  if ( age > 50 )
#   Message( first_name + ' ' + name + ' is older then 50');
```

This will cause the interpreter to stop before submitting the message.

Predefined breakpoints are ignored, when not running in debug-mode.

When running in debug mode, OSI automatically stops at the first statement to be executed.

### Break at error

When running in debug mode, the debugger will always stop, when an error has been detected. The command listed in the command line is the command that failed. You may correct data and repeat the command (continue), or you may skip the command (JumpOver).

### Breakpoint location

Breakpoints can be set at the beginning of a statement or block (if, for, switch, while). When setting a breakpoint for a while statement, OSI stops ones when entering the while statement the first time.

```
# while ( next )           // break ones
```

To break for each iteration in the loop, the breakpoint can be set within the while condition:

```
while ( # next )           // break always
```

This can be done also in a for statement. In a for statement, breakpoints can be set in front of each operand in the for statement:

```
for ( i=0; # i < age; I = i+1, # next )
```

This statement will break each time before checking the condition and always before selecting the next object instance in the collection (next).

You may set breakpoints in front of an if statement or within, but this does not make a big difference.

```
if ( # i < age )
```

```
# if ( i < age )
```

For CASE statements, you may break before checking the

case or after. Breakpoints for CASE statements can be set in front of the case statement and in front of the statements to be executed for the case, but not in front of the case operand.

#	CASE 5 : next;	// break befor check
	CASE 5 : # next;	// break when case is true
	CASE # a*b : next;	// SYNTAX error !!!!

The difference between the two variants is, that the first breaks before checking the CASE operand, while the second breaks only, when the CASE is true, i.e. the switch operand is 5 in this example.

### Debug commands

When a break point is reached, you will get a command prompt for entering debug commands. At each break point, the current statement is listed in the command line.



Besides the statement to be executed next, the debug prompt displays the current context, i.e. the collection or object instance the function has been called for (in the example above, this is the Persons collection).

Execution continues, when entering the run command (RUN).

Debug commands allow displaying variables and object data. A detailed description of debug commands is given in "Debug commands".

Besides specific debug commands you may enter most of the OShell commands described in "Database Utilities".

### Force halt

When running in a loop (e.g. after entering continue or run), the debugger may be forced to halt at next statement by pressing ctrl-c. When not running in debug mode, the debugger will be activated, supposed, the application is running as console application (e.g. by calling OShell or OSI). When running GUI applications, one may call code in order to run the application in a console window.

### Procedures

For running a series of debugger commands when reaching a breakpoint, you may define break point procedures. Breakpoint procedures for the current position can be invoked from within the debug prompt:






or by defining a breakpoint procedure when setting the breakpoint:

```
while ( #:bp1 next ) // break always
```

Setting breakpoint procedures causes the debugger to execute the commands in the breakpoint procedure each time, when the breakpoint is reached. Breakpoint procedures may contain any debug or OShell command except debug run-command, which will terminate the procedure.

Before setting a breakpoint procedure, the procedure must be loaded. This can be done from the debug prompt:



@bp1 must be a procedure defined in the loaded file. Details about the `LOAD` command are described in “Database Utilities/OShell”. Setting the procedure will, however, not check the name, because the procedure might be loaded later. When the procedure is not found, no commands are executed.

Another way of invoking breakpoint procedures is to load them in an OSI script file.

```
DEBUGPROCEDURES = c:\ODABA\debug.prc ;
```

`DEBUGPROCEDURES` statements cannot be defined within a class or function, i.e. they must be defined outside any statement, class, view or function.

You may insert any number of `DEBUGPROCEDURES` statements in your OSI file and in each included file.

## 11.2 Reload OSI functions

While running OSI applications in debug mode, one may automatically reload OSI function updated in the resource database. This allows correcting syntax errors when not properly checked in ClassEditor, setting breakpoints while running the application or updating code in order to remove bugs. The feature does not work for OSI functions loaded from external OSI library directory.

In order to automatically reload changes made in OSI code, the following option has to be set:

```
OSI.RELOAD=YES
```

### **Check functions**

After updating OSI functions in ClassEditor, those should be checked in any case in order to make sure, that last update timestamp has been set properly.

### **Confirm reload**

Usually, there is no confirmation requested, i.e. updated functions are reloaded when being called next time. Functions are not reloaded when already running, i.e. when calling a function recursively this will be reloaded after change only when calling the “top” function.

When detecting syntax error in an OSI function, the error is reported to console or application output area. In case of running in debug mode with console output, the application waits until correcting the problem and confirming the correction in the resource database.

## 11.3 Debug Commands

Debug commands include most of the OShell commands described in “Database Utilities/OShell”, except the Quit/Exit command. There are, however, some command extensions, which are available in the debug prompt, only.

Command names are not case sensitive. Some of the commands have abbreviations, but here, we refer to the long command names in most examples.

The list of commands is not complete. In order to get a complete command list, help may be called from the debugger’s command prompt.

### Run commands

A special subset of debug commands are run commands, which control the execution. Run commands will terminate the debug prompt and continue execution of the function.

Continue | c      The continue command executes the function(s) until reaching the next breakpoint.

Step | s          The step command breaks in the first statement of the called function or at the next statement, when the current statement is not a function call. Since some statements may have different breakpoints, the step-in command may break several times at the same line, when going through with the step-in command.

n                  The next command breaks at the next statement in the current function or at the next break point. Since some statements may have different breakpoints, the step-over command may break several times at the same line, when going through with the step-over command.

Finish | fi        The finish command finishes the current function and breaks at the next statement in the calling function or at the next break point.

JumpOver | o     The jump-over command will skip the execution of the next statement. Depending on the breakpoint setting for block statements it will skip the whole block, when the breakpoint reached is at the beginning of the block (e.g. before the while keyword) or just a block statement or operand, when the current breakpoint is within the block statement.

Run | r            The run command will continue execution without stopping at breakpoint anymore. The debugger will still break execution, when an error has been detected while

executing.

Quit | q This is a run command, which immediately terminates the application. When running the debugger under OShell, it does not terminate the OShell, but the debugger, only.

Exit The command will exit the OShell immediately. This is an emergency function, which might not close all databases properly.

Frame | f In order to inspect variables and statements in the calling function hierarchy, one may change the stack frame. The function allows changing the stack frame. The current stack frame number is usually 0.

Stack frame numbers are listed when calling back trace (bt). The currently selected stack frame is marked in the back trace list.

Jump | j Jumping allows continuing with a statement different from the currently listed. In order to change to a statement in a different frame, one has to change the frame before calling jump.

StackLimit | sl The stack limit might be set in order to detect errors resulting from recursive function calls. When a stack limit had been set, the OSI reports an error and breaks into debugger when the stack limit had been reached.

Stack limit may also be set by setting the option variable STACK\_LIMIT as ODABA option or environment variable.

## **Breakpoint commands**

Breakpoint commands allow setting and deleting breakpoints.

BreakAlways | ba The break-always command causes the debugger to stop at each statement, also when choosing 'continue' as next run command. Thus, break-always is similar to the step-in command.

The break-always command can be reset by using the run commands StepIn, StepOver, StepOut or JumpOver.

Break | b The break command allows setting a breakpoint at the current or passed line. In order to set break points in other frames than the current one, the frame command (f) has to be called for selecting the proper stack frame. Proper line numbers may be obtained by calling the list command in the selected frame (l).

The command allows defining a procedure entry point for the current breakpoint by passing an entry point name to

the command.

In order to set break points in different functions, one may set a break point by passing OSI function name.

**Break** myFunction 10

When the OSI function is not known in the current scope, the function name has to be preceded by class (and namespace) name.

**Break** "MyClass::myFunction" 10

When the function is stored in an external OSI file, the file has to be loaded before (load command).

In order to run a procedure containing debugging commands, a procedure name might be appended at the end of the break command

**Break** *proc\_name*

The procedure with the entry-point is called, when the breakpoint is reached the next time. When not yet being loaded, the procedure can be loaded explicitly by calling the "Load" OShell command.

When a breakpoint procedure has already been set, the command will deactivate the current breakpoint procedure and replace it by the new one. When not passing a procedure name, the current breakpoint procedure will be reset.

Disable | d

The disable command will reset the breakpoint at the current or passed line. In order to reset break points in other frames than the current one, the frame command (f) has to be called for selecting the proper stack frame.

In order to reset breakpoints in another function, the function name has to precede the line number:

**Disable** "MyClass::myFunction"10

## Display commands

Display commands allow displaying debug information about the current state of the debugger.

ListCurrent | lc

The command will list the current statement as being displayed when entering the debug prompt.

List | l

The command will list statements and line numbers for the function in the selected stack frame.

BackTrace | bt

The command will list the calling sequence until the

current position.

## **Data commands**

For displaying and changing data you may use all data commands, which are valid in OShell (see “Database Utilities/OShell”). Some commands, however, behave differently from the OShell commands.

Print | p

The command displays the attribute value for a variable or object. In extension to OShell, you may display not only object variables in the current context, but also parameter values and global or local variables defined in the function.

sav

The command allows changing the attribute value for a variable. In extension to OShell, you may modify not only object variables in the current context, but also parameter values and global or local variables defined in the function.

## 11.4 Debug functions

Several debug functions are provided for supporting offline debugging. Debug functions may be called from within OSI functions, from the OSI debugger command line, but also from within internal C++ functions.

<b>Debug functions</b>	Currently the following debug functions are available:.
DebugFunction	Displays the headline of the function currently active, or the headline for a function in the calling stack.
DebugLocals	Shows the names and values for local parameters
DebugParameters	Shows the names and values for all parameters
DebugVariable	Shows the value for a single parameter or variable.

## 12 Running OSI under OShell

You may run OSI scripts under OShell (see Database Utilities). Data sources referenced in the OSI script must be pre-selected or provided within the statement.

Now, you may enter an OSI function for a Persons collection:

### **begin or do**

OSI functions can be introduced by `osi do` or `osi begin`. Using `begin` keeps the result open after processing the query. This allows browsing complex results after evaluation.

When printing the result to console or to a file, it is suggested to use `do` instead.

### **comments**

In order to add comments, those have to be defined on separate lines and beginning with `//` at the beginning of the line.

### **Debug mode**

In order to enable or disable the script debug mode, the `OSI.DEBUG` option might be set at the beginning of the OShell procedure.



## 13 References

- [1] ODMG: *The Object Data Standard ODMG 3.0*, Academic Press, 2000
- [2] ODABA Script Language Reference, 2005
- [3] ODL Guide, 2005
- [4] R. Karge, *Unified Database Theory*, run-software, 2003