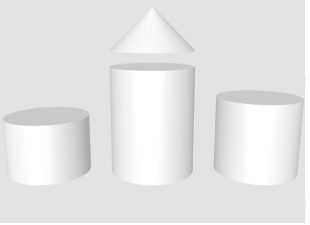


01101001001110010101  
10101101010010111011  
10001011101010101011  
10110010100101011010  
10101001101011010010  
01110010101101011010  
10010111011100010111



**ODABA<sup>NG</sup>**

01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101

**Data Exchange**

0 11 01  
0 1010 01  
1 11101 01  
0 10101

n

00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
1101100101001101101  
01010100110011010010

0101011011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110011010010  
01110010101101011010  
10010111011100010111  
01010101011101100101  
00101011010101010011  
00110100100111001010  
11010110101001011101  
11000101110101010101  
11011001010010101101  
01010100110



**run Software-Werkstatt GmbH**  
**Köpenicker Strasse 325**  
**12555 Berlin**

www.run-software.com  
Tel: +49 (30) 65762791  
e-mail: run@run-software.com

Berlin, August 2010

# Content

- 1 Introduction.....5**
  - ODABANG.....5
  - Platforms.....5
  - Interfaces.....5
  - User Interfaces.....5
  
- 2 Data Exchange.....7**
  - Command line Tools.....7
  - GUI Tool.....7
  - OSI expressions.....8
  - PropertyHandle.....8
  
- 3 Data exchange definition.....10**
  - Access functions.....10
  - File access parameter.....10
  
- 4 Data Exchange schema.....13**
  - File Schema.....13
    - File schema.....13
    - Dictionary.....15
    - ODL.....15
    - OXML schema.....15
    - CSV/ESDF.....15
    - Delimiters.....17
  
- 5 External file formats.....19**
  - BINA.....19
  - ESDF, CSV.....19
  - OIF.....20
  - OXML.....21
  - Accessing external files.....21
  - PropertyHandle.....22
  - Running GUI Index Services.....24
  - Predefined settings.....25
  - Indexing objects.....26
  
- 6 Console Index Services.....29**
  - Ini-file.....29
  - Maintenance options.....30
  - Running console Indexing Services.....31
  
- 7 Index Manager Options.....33**
  - collection.....33
  - field1...9.....33
  - Switches.....34

<b>8 Create application specific Index Services.....</b>	<b>36</b>
IndexManager class.....	36
Opening IndexManager.....	36
Indexing process.....	36
Keyword search.....	37

# 1 Introduction

## ODABA<sup>NG</sup>

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

ODABA<sup>NG</sup> 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<sup>NG</sup> 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<sup>NG</sup> applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

## Platforms

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

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

## Interfaces

ODABA<sup>NG</sup> supports several technical interfaces:

- C++, COM as application program interface (this allows e.g. using ODABA<sup>NG</sup> 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<sup>NG</sup> provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA<sup>NG</sup> provides a special ODABA<sup>NG</sup> GUI

builder.

## 2 Data Exchange

Data Exchange provides different ways of importing or exporting data from an ODABA<sup>NG</sup> database to extended comma separated files (ESDF, CSV), to xml files (OXML, XML) or to object interchange format files (OIF). OIF is the proposed standard format for exchanging data between object-oriented databases (ODMG).

While the capabilities of ESDF (CSV) are limited, OXML and OIF allow transferring the complete content of a database. Even though XML is the more common format, OIF has the advantage that it should be supported by all object oriented databases and it consumes less space.

Besides different file formats ODABA<sup>NG</sup> provides different data exchange technologies.

### Command line Tools

ODABA<sup>NG</sup> provides two data exchange tools, one for import (**Import**) and another for exporting data (**Export**).

#### Import

Import provides features for importing a file with a valid import format into an ODABA database. This is a preferred way for importing data periodically, in which case a batch job can be prepared and called whenever required.

It is a possible but not the most comfortable way for ad-hoc import processes, which can be solved better with the GUI Data Exchange or from within OSI programs.

#### Export

Import provides features for exporting selected data from an ODABA<sup>NG</sup> database to a file with one of the defined formats. Also, this is a preferred way for exporting data periodically, while ad-hoc data export becomes more comfortable from within OSI or by using the GUI Data Exchange tool.

### GUI Tool

The Data Exchange GUI tool provides features for designing the content of a data exchange and running the data exchange directly or creating a data exchange definition file (data exchange schema).

The data exchange schema can be referred to later when calling the command line tool or within an OSI script.

The GUI tool provides the most comfortable way for

designing a data exchange schema. It allows also running the defined data exchange (e.g. for testing purpose).

## **OSI expressions**

In many cases data exchange is simple and can be directly called from within an OSI expression. OSI OQL provides two built-in functions in order to import and export data. Those functions are the same functions which are called from the command line and the GUI tool.

### ToFile

ToFile writes data from a defined collection to an external file with one of the defined formats. The OSI query may create a view for the data to be exported. Since the data exchange schema supports property selections, this is, however, not necessary in most cases.

### FromFile

FromFile imports data from an external file with one of the supported formats into a collection. In general, one cannot import data into a view, but there are views, which are partially updateable, which would allow importing data as well.

## **PropertyHandle**

You may access an external file by property handle. This allows reading or writing data from a program or from within an OSI expression. Property handles for external files will not, however, import or export data automatically.

Opening a file via property handle activates the rich property handle functionality for the external file. Although there are many features, which cannot be supported for an external file, many helpful functions of property handle are still working for this data source type,

Accessing external data via property handle does not require an exchange schema. A file schema, which does not define data mapping, would be sufficient. Since file schemata for CSV files can be derived very simple in many cases, the external file does not require additional information for being accessed.

The property handle access functionality is the base for the OSI functions FromFile and ToFile.

### Open

The file schema for external files can be defined in advance within the ODABA<sup>NG</sup> dictionary as structure and extent definition. In this case, the external file can simple

be accessed via the extent name, similar to any other extent in the database.

#### OpenExtern

Often, it is not very comfortable defining structure and Property handles for external files in the dictionary. Especially CSV files carry metadata in the headline, which contains sufficient information for extracting a file schema. Thus, property handle support an additional function for opening external data sources, which are not defined in the dictionary. This allows accessing data ad-hoc and in much simpler in many cases.

### 3 Data exchange definition

Data exchange definitions describe the file data source, the schema location and format types for exchange file and schema. Usually, the exchange definition is specified in a **ToFile**, **File** or **FromFile** operation, but this might be hidden behind a more comfortable user interface.

#### Access functions

External files can be accessed in different ways.

- File – Read or write explicitly
- FromFile – Import from file
- ToFile – Export to file

**File** The File() function allows accessing an external file structure, which is defined by an explicit or implicit file schema. External files can be read or written, but depending on the file structure, there are several restrictions. Most external file formats do support appending data to the file, only.

**FromFile** The FromFile() function supports importing data from external files into a database. Importing files requires a (usually explicit) data exchange schema (extended file schema), which provides a mapping to database locations in addition to the structure definition of the import file.

**ToFile** The ToFile function supports exporting data from a database to an external file format. Es well as the FromFile() function, ToFile requires a data exchange schema.

#### File access parameter

All file functions refer to same set of parameters, which describe the location for data and file or exchange schema.

```
file           := 'File' foperand_list
from_file     := 'FromFile' foperand_list
to_file       := 'ToFile' foperand_list
foperand_list := 'Path' '=' string [foptions(*)]
foptions      := ',' foption
foption       := file_type | file_schema | headline
file_type     := 'FileType' '=' type_name
type_name     := 'ODL' | 'OXML' | 'OIF' | 'CSV' | 'ESDF' |
               'BINA'
file_schema   := 'Definition' '=' def_location,
def_location  := structure_name | string
```

```
headline := 'Headline' '=' boolean
```

At least the file path must be passed as operand to the file access functions. Additional file options can be passed for providing file and exchange schema and file type.

**file\_type** ODABA supports different external file types. The file type need not to be defined, when the file name passed in Path has one of the following extensions:

- BINA - binary flat file (.bina)
- CSV, ESDF - extended self delimiter file (.esdf, .csv)
- OXML - ODABA xml file (.oxml, .xml)
- OIF - object interchange format (.oif)

**file\_schema** The file or exchange schema can be provided together with the data. When the file or exchange schema is passed in a separate file, the Definition option refers to the location of the file definition. When no separate file schema is passed the file schema is supposed to be part of the external file (e.g. headline in an ESDF file).

When the file or exchange schema is passed with the data file (no file schema), the definition format has to correspond to the format of the data file.

- BINA – no file definition supported in the file
- CSV, ESDF – ESDF headline format
- OXML - ODABA xsd definition
- OIF – ODL definition

**def\_location** The file or exchange schema can be provided as definition in a dictionary, in which case the *structure\_name* refers to a structure definition in the dictionary. The file or exchange schema might also be provided in a separate file as ODL (.odl), OXML (.oxsd) or ESDF (.esdf) definition files, in which case the location is passed as quoted string pointing to the file location.

The system determines the proper type from the file extension. When no valid extension could be found, the system tries to analyze the definition file type by file content:

first character '<' : OXML format

first character '{' or beginning with a word followed by a separator : ESDF format

Beginning with **schema** keyword, ODL is assumed.

## Headline

The headline option indicates, whether the external data file contains an imbedded file schema (typically the headline in CSV or ESDF files). Either headline or schema location must be provided in order to obtain the file schema for input operation.

In case of output operation, schema definition in the output file header will be ignored, i.e. the exchange schema must be defined in a separate definition (database schema or schema file). When no exchange schema has been provided, the input structure is used as an implicit exchange schema, i.e. all attributes and depending object instances are exported to the output.

When defining both, the schema location is used. In some cases, the schema location is verified against the headline definition, in this case.

## 4 Data Exchange schema

A data exchange schema is required for any type of data exchange in order to provide the mapping rules between internal and external data. The data exchange schema is an intensional schema, i.e. it refers to structure definitions, only. Thus, a data exchange schema can apply on any collection (database) or file (external data source), which fits into the rules defined in the data exchange schema.

Data exchange schemata can be provided in different formats. The format of the data exchange schema does not depend on the file format for the external data source. Thus, you may still use the same data exchange schema definition, even though you have changed the format of the external file. Data exchange schemata can be provided in one of the following formats:

- Dictionary – Structure definition in an ODABA dictionary
- CSV/ESDF – Headline definition format
- OSI ODL – Schema definition language
- OXML – extended XML schema definition

The data exchange schema is an extended file schema with additional mapping rules for assigning external data fields to database properties. The database property correspondence is always defined in the source attribute, which is an extension for all file schemata.

## File Schema

### File schema

The file schema contains the structure or data type definition(s) required for describing the data in the external file. Most of the rules for defining schemata of types mentioned above are described in other documents. Thus, only specific rules to be taken into consideration when providing a file schema will be described here.

Since all supported file formats are hierarchical formats, i.e. properties or fields may contain sub-properties or

collection of related instances. There are limitations in a few cases (e.g. for binary files), but this is no contradiction for providing common principles when defining a file schema.

Common file schema

In contrast to database schema (object model), the file schema does not support orders and relationships. The following BNF definition provides an idea of the common definition elements provided in all definition formats.

```
record      := field_list
field_list := field(*)
field      := [name] [data_type] [size]
           [sub_fields] [dimension] [db_source]
sub_fields := field_list
```

**record** Even though it be confusing to speak about a record in a hierarchical data structure, we will the term record as entry for the definition, since in many cases, one knows exactly what a record is. Sending Person data might be as complex as possible, but we will probably consider data for each person as a record in this data set.

**name** A record (or structure) consists of a number of fields (properties, attributes). Each field may have got a name (if not, artificial names are created as *field0001*, *field0002* etc.).

**data\_type** The default data type is STRING (except for binary files, which may contain binary data as well).

**sub\_fields** When a field (or field Instances) are structured, a list of sub-fields (describing a structure again) can be defined. Each field in the sub-field list may have sub-fields etc.

**dimension** The default dimension is 1. Any other positive number for fixed arrays or 0 for collections with undefined number of elements may replace the default dimension.

**db\_source** The database source defines the corresponding data source in a database. This might be a property name or path, but not an expression. A data source is required for importing or exporting data from/to external files but not for reading external files by property handle, only.

Each schema supporting these requirements is able to describe a file schema. It becomes obvious, that OXML schema and ODL provide these requirements, as well as the ODABA dictionary does. For CSV or ESDF a specific file definition format has been defined, which, in the simple case, corresponds to the CSV head line.

## Dictionary

Describing an external file structure in the dictionary might be the most comfortable way for complex data structures. Dictionary structures for external files may consist of attributes, references and exclusive base structures. External file definitions must not contain relationships.

For assigning a data source to a field in an external file is possible by means of the property *source* reference. In case of defining more than one source references for a property the assignment is done by field name, which must be assigned as source definition name in this case.

## ODL

The ODL schema definition is a script equivalent to the dictionary definition. It follows the same rules as defining a structure in the dictionary. The example below shows a complete definition for a Person data exchange file.

```
STRUCT XAddress {
    STRING    f_zip           SOURCE(zip);
    STRING    f_city         SOURCE(city);
    STRING    f_street       SOURCE(street);
    STRING    f_number       SOURCE(number);
};

CLASS XPerson {
    ATTRIBUTE {
        STRING    f_pid           SOURCE(pid);
        STRING    f_name          SOURCE(name);
        STRING    f_first_name    SOURCE(first_name);
        STRING    f_birth_data    SOURCE(birth_date);
        STRING    f_sex           SOURCE(sex);
        STRING    f_married       SOURCE(married);
        STRING    f_income        SOURCE(income);
    };
    REFERENCE XAddress f_location[3] SOURCE(location);
};
```

Depending on import or export functions the database source acts as target or source.

## OXML schema

An OXML schema is another equivalent for a dictionary structure definition and can be used instead of an ODL or dictionary definition.

## CSV/ESDF

The definition for CSV or ESDF (Extended Self Delimiter Files) is an extension of a CSV file headline. In the minimal case it only consists of variable names.

In order to support more complex data structures in a comfortable and CSV compatible format, we introduced

ESDF, which is a CSV extension, since it supports complex attributes as well as references.

The rules for defining a CSV or ESDF file are described in the subsequent BNF definition.

```
Headline      := fields
fields        := field [ field_ext(*) ]
field_ext     := sep field
field         := [name] [size] [sub_fields] [dimension]
[source]
source        := '=' path
size          := '(' number ')'
dimension     := '[' number ']'
sub_fields    := '{' fields '}'
path          := path_element [ path_extension(*) ]
path_extension:= '.' path_element
path_element  := name [ parameter ]
parameter     := get_parm | provide_parm
get_parm      := '(' value ')'
provide_parm  := '[' value ']'
value         := path | constant

Data          := items
items         := [ item ] [ item_ext(*) ]
item_ext      := sep [ item ]
item          := dvalue | item_set | item_block
item_set      := '[' items ']'
item_block    := '{' items '}'

sep           := ';' | '|' | '\t'
```

The BNF describes the ESDF header and the data lines. In contrast to CSV, ESDF limits field delimiter to ‘;’, tab and ‘|’. Undefined symbols *name*, *string*, *dvalue* and *constant* are standard symbols and do have the following meaning.

*name* Is a field name which usually starts with an alphabetic character or underscore.

*number* Is an integer value.

*dvalue* Any sequence of characters not containing field, string, instance, collection or line separators (see “Delimiters” below)..

The file definition for an ESDF file is usually passed in the first line of the file (headline). I might be passed, however, also separately from the data file.

```
f_pid = pid; fname = name; f_first_name = first_name;
```

```
f_birth_date = birth_date; f_sex = sex; f_married = married;
f_income = income; f_location {f_zip = zip; f_city = city;
f_street = street; f_number = number}[3] = location
```

When names in the headline are identical with database source names, source assignments can be omitted:

```
pid,name;first_name,birth_date;sex;married;income;location{zip
; city;street;number}[3]
```

When defining the file definition separately instead of providing a headline, the definition may contain line breaks:

```
f_pid      = pid;
fname      = name;
f_first_name = first_name;
f_birth_date = birth_date;
f_sex      = sex;
f_married  = married;
f_income   = income;
f_location {
  f_zip     = zip;
  f_city    = city;
  f_street  = street;
  f_number  = number
} [3]      = location
```

## Delimiters

ESDF defines a reserved set of delimiter characters. Delimiter characters must not appear in values without being quoted.

**Field delimiter** Characters ‘;’, ‘|’ and ‘\t’ (tab) are considered as field delimiter. Field delimiters are considered as such, also when appearing mixed, i.e. also when creating an ESDF file using ‘\t’ as field separator, values containing a ‘;’ must be enclosed in string delimiters.

**String delimiters** ‘”’ and ‘”’ are considered as string delimiters. The starting string delimiter must be the terminating delimiter, too. Starting a string value with ‘”’, the value may contain ‘”’ and reverse. When starting string delimiters need to be coded within the string, those must be preceded by an ‘\’.

```
'my name is "Paul"' // valid
'my name is \"Paul\"' // valid, same as above
'my name is \'Paul\'' // valid
'my name is `Paul`' // valid, same as above
```

Instance delimiters

Instance delimiters '{' and '}' are used to define begin and end of complex (structured) data values. Instance delimiter may appear within value collections but also outside collections. Instance delimiters are not required for base structure members.

Collection delimiter

Collection delimiters '[' and ']' are used to define value or instance collections.

## 5 External file formats

ODABA supports different external file formats, which can be accessed directly via PropertyHandle access functions or via file functions File(), ToFile() and FromFile.

### BINA

Binary files are files with a fixed data structure and can be considered as the most compressed format for data exchange.

There are, however, several limitations in using binary files.

- Binary files always require an external file definition (no headline definition supported).
- Binary files do support arrays or references with fixed number of elements, only.

In contrast to other files formats, binary files may contain integer and float values or other binary data types.

### ESDF, CSV

The Extended Self Delimiter File format is an extension of the CSV format. ESDF files contain one record per line, i.e. line break indicated the end of a record. In contrast to CSV, ESDF supports complex attributes and references.

Since ESDF does not require any tags, it is the most efficient way of exchanging large data files. On the other hand, it requires fields being defined in a correct sequence.

#### Specification

ESDF has a simple BNF specification as described below:

```
ESDFFile      := [ header ] esdf_record(*)
Header        := Headline nl
esdf_record   := Data nl
```

As line break, new line (NL), carriage return (CR) or both are accepted after headline and between data lines. Headlines are optional. File definitions might be also passed separately.

#### Headline

ESDF files may contain a headline defining the file or exchange schema. Since headlines need not differ syntactically from data lines, the file definition must pass the headline option in order to indicate, that an ESDF file

contains a headline.

## OIF

The object interchange format is a standard suggested by ODMG. The ODABA OIF has some extensions and some limitations compared with the ODMF OIF. Nevertheless, there is a big common denominator between both, which makes it possible exchanging data in OIF format with any other object-oriented database supporting OIF.

In contrast to other external file formats, OIF supports additional features as the distinction between creating and overwriting data in the database during import.

## Specification

An OIF file is an alternate recursion between property and instance values. Each property value may consist of a number of instance values and each instance value consists of one or more property values. Thus, an OIF file may contain a number of instances, but also a collection (property), which contains a number of instances.

```
OIF                := OIFData | OIFInit
OIFData            := prop_init(*)
OIFInit            := prop_list | inst_list

prop_init          := identifier ['='] prop_value [' ','']
prop_value         := inst_init | inst_list
inst_list          := '{' inst_init(*) '}'

inst_init          := [ inst_intro ] [ locator ] inst_value
[' ','']
inst_intro         := [ identifier ] scoped_name
locator           := update_locator | create_locator
update_locator    := '(' loc_init ')'
create_locator    := '[' loc_init ']'
loc_init          := constant | prop_init(*)
inst_value        := constant | prop_list
prop_list         := '{' prop_init(*) '}'
```

## Delimiters

Value delimiter ',' and assignment operator are optional and should not be used when compatibility is required.

## Locators

In order to distinguish between replacing or creating, ODABA OIF supports create and update locators. Create locators follow the standard and will create new instances when not yet existing.

In contrast to ODMG OIF, which supports numerical locators, only, ODABA OIF supports key locators, as well. When passing a number in *loc\_init*, this is

interpreted as position in a collection or in an array. When passing a string, it is interpreted as key value. Component key values can be defined either by passing a string value with component values separated by '[' ( [ 'Miller|Paul' ] ) or by passing the values by property names ( [ first\_name 'Paul', name 'Miller' ] ).

**Property lists** ODMG OIF supports property values by position, i.e. without preceding property name. Since this implies a high risk for value mismatch, ODABA OIF does not support this feature and requires a property name in front of each value assignment.

**Head line** When defining a "headline" at the beginning of an OIF file, this must be defined as ODL schema definition beginning with the **schema** keyword.

```
SCHEMA {  
    ... // file schema definitions  
}
```

**OXML** OXML is an xml format with several ODABA specific schema extensions. Thus, xml is able to reflect the complexity of ODABA database object model definition completely.

Providing an OXML schema separately, xml files can be accessed via an OXML dictionary as OXML database.

Using file access for accessing xml data, however, allows importing or exporting xml data directly from/to an external file according to the database source definitions. Moreover, an xml file can be describes using an ODL definition, which might be more comfortable.

## Accessing external files

There are different ways for accessing external files. External files can be accessed from within a program using the PropertyHandle function OpenExtern(). Another way is accessing external files via OSI scripts using File(), FromFile() or ToFile() functions.

It is also possible defining extents in the database referring to external files. In this case, the external file can be accessed simply by opening the extent with

appropriate PropertyHandle functions.

## PropertyHandle

PropertyHandle for external files can be created two ways. One is defining an external extent in the dictionary in advance. The other is to open a property handle calling OpenExtern().

### External extent

When defining an extent for an external file, this can be accessed by property handle functions after creating an appropriate property handle.

Defining an extent for accessing allows defining one or more sort orders (indexes) for the extent, which are created when opening the extent.

```
PropertyHandle ph(obhandle, "ExtFile", PI_Read);
```

After opening the property handle simple property handle access functions (Get, Position, NextKey etc.) can be used for selecting instances in the external extent.

### OpenExtern

OpenExtern() allows opening a property handle for external file access without defining it in the dictionary in advance.

```
PropertyHandle ph;  
char *path = "externalFile.esdf";  
char *filetype = NULL; // ESDF from extension  
char *definition = NULL; // definition in headline  
  
ph.OpenExtern(obhandle, path, definition, filetype, PI_Read);
```

After opening the property handle simple property handle access functions (Get, Position) can be used for selecting instances in the external extent.

OpenExtern() provides access to the external file but does not automatically import or export the file.

## [SYSTEM]

DICTIONARY=C:\odaba\adk.sys

## [ODE90]

RESOURCES=RESSECT

DATA=DATSECT

PROJECT=IndexServices

PROJECT\_DLL=Designer

CTXI\_DLL=AdkCtxi

DESIGNER\_RES=C:\odaba\res

DSC\_Language=English

## [RESSECT]

DICTIONARY=C:\odaba\adk.sys  
DATABASE=C:\odaba\adk.dev  
NET=YES  
ONLINE\_VERSION=YES

**[DATSECT]**

DICTIONARY=C:\odaba\adk.sys  
DATABASE=I:\opa\opa.dev  
NET=YES  
ONLINE\_VERSION=YES  
ACCESS\_MODE=Write

**[IndexManager]**

keywords=DSC\_Keyword  
stopwords=DSC\_Stopword  
lexterms=DSC\_LexTerm

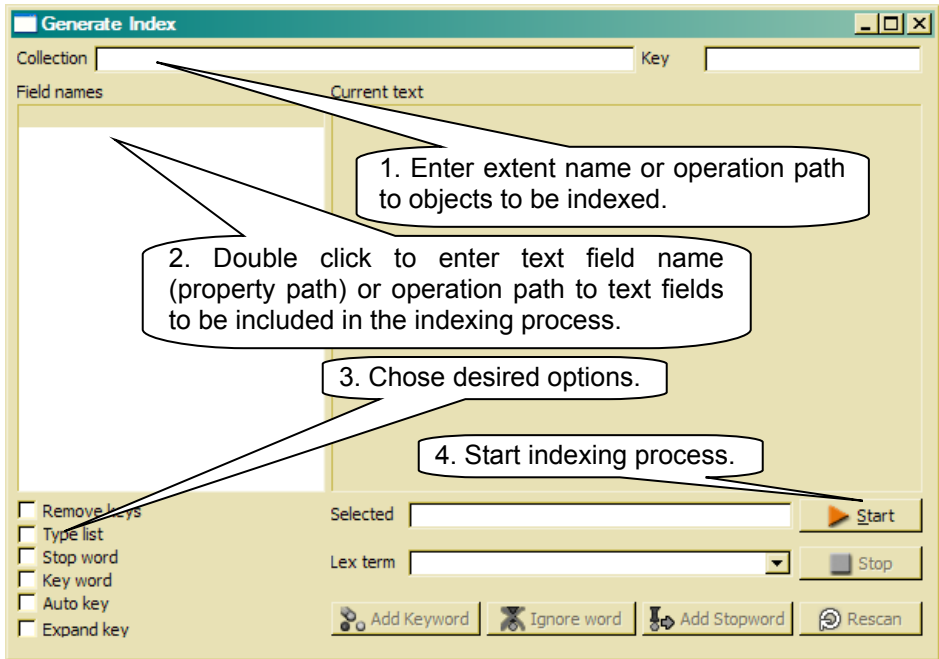
- [SYSTEM] The system section refers to database system information. The minimum required is the DICTIONARY reference to the system dictionary, which is stored in the ODABA<sup>NG</sup> installation folder. When running the application with a system dictionary stored on the server, server name and a port number have to be defined as well.
- [ODE90] The ODE90 section contains information for the ODABA<sup>NG</sup> GUI runtime environment. It refers to sections for resource database and database locations and contains some details for the Index Services application. This section must not be changed.
- [RESSECT] This section defines the connection to the application resource database, which is the adk.dev database provided on the ODABA<sup>NG</sup> installation folder. This section must be updated, when ODABA<sup>NG</sup> had been installed on a different location as the default location or when running the application in a Unix or Linux environment.
- [DATSECT] This data section defines the connection to the application database by defining the dictionary and the database. When indexing a resource database (as in the example above), the dictionary is the system dictionary adk.sys provided in the ODABA<sup>NG</sup> installation folder.
- Usually, paths for dictionary and database must be replaced by the application database (DATABASE) and the application resource database (DICTIONARY).

[IndexManager]

The Index Manager section defines the collection names for keyword, stop-word and lexical base term collections. Usually, one refers to the default collections as in the example above. Sometimes, it becomes necessary to define different keyword collections for different indexing processes. In this case, additional keyword, stop-word and lexical base term extents must be defined in the application resource database before being referenced here.

### Running GUI Index Services

When calling Index services with this type of minimum configuration, an empty application appears:



Defining object collection

You may enter an operation path to the object collection to be indexed in the **collection** field.

Defining text fields

After defining the object collection to be indexes, you can chose up to 10 text fields or operation paths to text fields to be evaluated by the indexing process. Text fields must by valid properties in the context of the object type for the selected object collection.

In order to get a list of available text fields, enter \* in the first list line and press the **Start** button. Then, a list with the maximum 10 text fields (properties) defined for the object type will be displayed. You may remove text fields by using the **Remove** function from the context menu. Using the Insert function from the context menu will insert an empty line for entering another text field in the list.

For defining additional text fields, you may also enter text field names or operation paths into empty lines at the end of the list.

Selecting index options

Desired options can be switched on in the option list. The meaning and affect of those options is described in "Index options".

Start indexing process

Finally, you may press the **Start** button to run the indexing process with the current settings.

### Predefined settings

In order to simplify running an index service, you may provide extended Index Services settings in the configuration or ini-file in the [IndexManager] section:

...

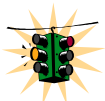
#### [IndexManager]

```
keywords=DSC_Keyword
stopwords=DSC_Stopword
lexterms=DSC_LexTerm

collection=NamedTopics.OrderBy(sk_ident)

field1=definition.name
field2=definition.definition.characteristic
field3=sub_topics().definition.name
field4=sub_topics().definition.definition.characteristic
field5=definition.lable
field6=sys_ident

stop_word=YES
remove_keys=NO
type_list=YES
```



collection

Note, that option variables are case sensitive and no spaces are allowed between name and '=' sign, when using an ini-file as in the example above. Spaces can be inserted when using a configuration file (xml) instead.

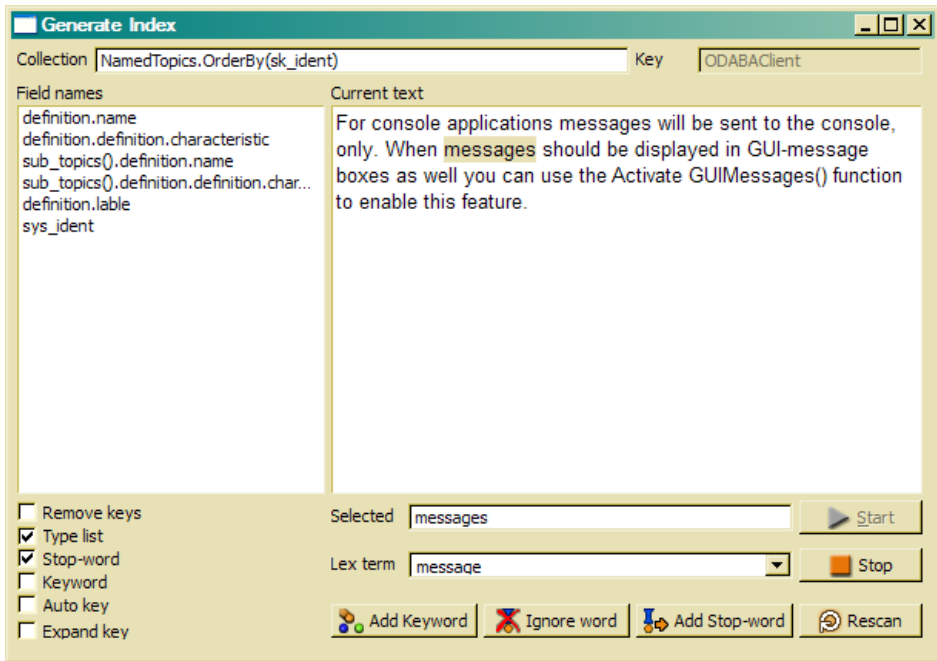
In the example above, the operation path to the collection changes the sort order, which helps seeing the progress.

Text fields      The configuration or ini-file allows defining up to 9 text fields, only, and not 10, as possible in the GUI application.

Field 3 and 4 refer to a collection of subordinated text fields, which are included into the evaluation as well.

Options      All options defined in the GUI tool can be defined in the configuration or ini-file. Usually, you need to define only those fields, which are to be enabled. User-defined index applications, however, may refer to more than one indexing process. Hence it is more save, always to define all options in the [IndexManager] section.

**Indexing objects**      After the options have been set properly, you may start the index processing by pressing the Start button.



The Index Manager associates each selected object with all keywords found in the listed text fields. In the example above, this means, that objects (Topics) are also associated with keywords found in related object instances (sub\_topics), as defined in field 3 and 4.

Index services will stop, when a word found in the text is not defined as keyword or as stop-word. The critical word is highlighted in the text and displayed in the **Selected** field below the text box.

Now, you can decide, whether the word found is a keyword or stop-word.

Create stop-word

When the word selected has been identified as stop-word, click the **Add Stop-word** button. The selected word will be added to the stop-word collection and not be questioned any more, supposed the **Stop-word** option is switched on.

Create keyword

When you decide, that the current word is a keyword, a lexical base term should be selected from the drop-list below or entered in the **Lex term** field. The lexical base collects all keywords with the same meaning. This allows finding a text which might contain the word 'properties' when searching for 'property'.

Theoretically, the word used for the lexical base term does not matter, but practically is helps much using a sort lexical normalized word form. The Index Manager tries to locate a lexical base term when detecting a new word and displays it in the **Lex term** field.

When you do not want to create a lexical base term, the **Lex term** field must be empty before adding the keyword.

For creating a new keyword, you just click on the **Add keyword** button.

Ignore word

When the current word is neither a keyword nor a stop-word, you may ignore the word by clicking **Ignore word**.

Spelling correction

When the selected word is just misspelled text, you may correct the highlighted text in the text box above. After changing the text in the text box, we suggest to press the **Rescan** button in order to include the word changed in the indexing process.

Changing options

During the indexing process, you may change the options at any time. Thus, you may switch on the **Keyword** option in order to continue indexing based on the keyword collection defined so far.

Terminate process

In order to terminate the indexing process, you may press the Stop button, which allows you starting a new indexing process. You may also leave the application by clicking on the close button (**x**) in the upper right corner of the application form.

Progress indicator

In order to get a slight idea about the progress of the indexing process, the key of the currently selected instance is displayed in the **Key** field above the text field.

## 6 Console Index Services

For running the Console Index Services, you need to prepare a configuration or ini-file, which contains all required information for the indexing process. With that configuration file you may call the Index Services as:

ODABA/IndexServices.exe *ini-file*

### Ini-file

The configuration or ini-file contains the definitions for the data sources, object collections to be indexed and text fields.

#### [SYSTEM]

DICTIONARY=C:\odaba\adk.sys

#### [IndexServices]

DICTIONARY=C:\odaba\adk.sys

DATABASE=!:lopalopa.dev

NET=YES

ONLINE\_VERSION=YES

ACCESS\_MODE=Write

DSC\_Language=English

#### [IndexManager]

keywords=DSC\_Keyword

stopwords=DSC\_Stopword

lexterms=DSC\_LexTerm

collection=NamedTopics

field1=definition.name

field2=definition.definition.characteristic

field3=sub\_topics().definition.name

field4=sub\_topics().definition.definition.characteristic

field5=definition.lable

field6=sys\_ident

stop\_word=YES

remove\_keys=YES

type\_list=YES

auto\_key=YES

#### [SYSTEM]

The system section refers to database system information. The minimum required is the DICTIONARY reference to the system dictionary, which is stored in the ODABA<sup>NG</sup> installation folder.

[IndexServices] This IndexServices section mainly defines the connection to the application database by defining the dictionary and the database. In the example above the dictionary is the system dictionary adk.sys provided in the ODABANG installation folder, but it might be also an application resource database, when going to index object instances in an application database.

Usually, paths for dictionary and database must be replaced by the application database (DATABASE) and the application resource database (DICTIONARY).

In addition the section defines some application settings for the Index Services, as e.g. the language (DSC\_Language).

[IndexManager] The Index Manager section defines the collection names for keyword, stop-word and lexical base term collections. Usually, one refers to the default collections as in the example above.

## Maintenance options

In principle, it is possible to run console Index Services for building indexes and keyword collections as described for the GUI Index Services. But the basic idea is to run cyclic maintenance processes in order to update the object/keyword associations.

Best matching Typically settings or maintenance applications are the following options:

```
stop_word=YES  
remove_keys=YES  
type_list=YES  
auto_key=YES
```

With this configuration, stop-words are checked and keywords are automatically created, when not yet being defined as keyword or stopword. Newly created keywords are not associated with lexical base terms.

This provides best matching results after maintenance, but quality is not as good, since different word forms are not recognized as same.

Best quality Alternatively, maintenance can be called with the following options:

```
stop_word=YES
remove_keys=YES
type_list=YES
key_word=YES
```

In this case, unknown words will be ignored and object instances are associated with known keywords, only. This allows calling GUI Index Services later on (e.g. once a week or once a month) in order to assign new keywords and lexical base terms manually.

This maintenance type does not provide good matching results, since new keywords cannot be searched. After running the GUI Index Services for creating new keywords, the quality is better.

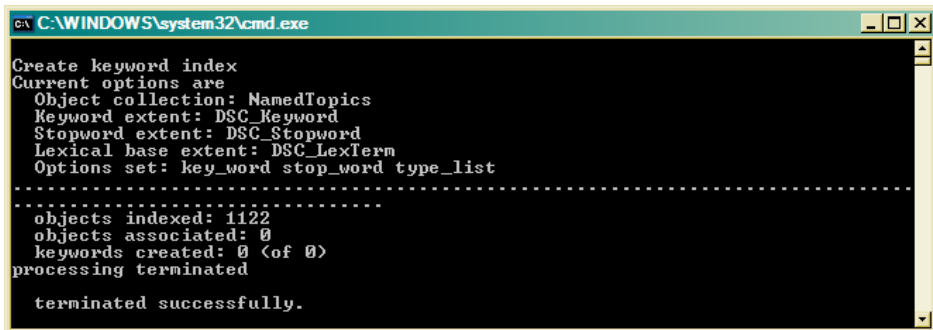
Optimal solution The optimal solution would be to run a sort of auto-matching between new keywords and lexical base terms. Still, we will need a good tool for maintaining mismatches and turning keywords into stop-words.

We are working on better solutions and waiting for your comments.

### Running console Indexing Services

When calling Index Services with a maintenance configuration as described above, the indexing process runs without user interaction.

The options are displayed on the console and the number of indexed objects and keywords created is displayed at the end of the session.



```
C:\WINDOWS\system32\cmd.exe
Create keyword index
Current options are
Object collection: NamedTopics
Keyword extent: DSC_Keyword
Stopword extent: DSC_Stopword
Lexical base extent: DSC_LexTerm
Options set: key_word stop_word type_list
-----
objects indexed: 1122
objects associated: 0
keywords created: 0 (of 0)
processing terminated
terminated successfully.
```

When running console Index Services without key\_word and auto\_key option, the process stops at the first unknown word.

```
C:\WINDOWS\system32\cmd.exe
Create keyword index
Current options are
  Object collection: NamedTopics
  Keyword extent: DSC_Keyword
  Stopword extent: DSC_Stopword
  Lexical base extent: DSC_LexTerm
  Options set: stop_word type_list
.base - word not found (enter 'k', 's', 'i', 'c' or '?')?
k - create keyword
s - create stopword
i - ignore word
c - cancel process
>
```

The console Index Services let you decide between defining a keyword or a stop-word. You may also ignore the word currently selected, but you cannot assign a lexical base term to the word.

This is, however, a simple way to estimate the density of unknown keywords in the system, which is a measure for running manual keyword maintenance in order to improve the index quality.

## 7 Index Manager Options

This is a short summary of settings for the Index Manager, which is usually called by the Index Services but could also be called by user-defined indexing processes.

**collection** The collection option defines the path to the object instances to be indexed.

```
collection=NamedTopics
```

In simple cases this is an extent name, but it could be a more complicated operation path, as well.

```
collection=NamedTopics().sub_topics
```

With the last collection definition, all sub-topics could be indexed as individual object instances. Thus, they become accessible via a keyword index.

Since objects to be indexed need an object identity (LOID), the collection path must not refer to transient object instances (i.e. in view)

```
collection=NamedTopics().Select(title = definition.name,  
text = definition.definition.characteristic)
```

The example above is not valid, since the select operator creates transient instances, which cannot be indexed.



Note, that option definitions in ini-files must not have line breaks. Using a configuration file might be more comfortable, but here '<' and '>' must be coded as &lt; and &gt;.

**field1...9** Field options provide text fields or properties defined in the structure of the collection selected by the collection path.

Fields may refer simply to text properties in the object instance:

```
field1=definition.name  
field2=definition.definition.characteristic
```

In some cases text properties in subordinated objects conceptually count as object properties. Thus, field options may also refer to collections of text fields, by defining operation paths.

```
field3=sub_topics().definition.name
field4=sub_topics().definition.definition.characteristic
```

Here, the name and characteristic field from all related sub-topics are included in the indexing process.

Field definitions may also define views, since text fields act as criteria for associating the object instance with a keyword, only, and are not referenced physically.

## Switches

Switches or Index Manager options allow controlling different indexing strategies.

### stop\_word

When the stop-word option is switched off, stop-words will not be checked. In case a stop-word is also stored in the keyword collection, this allows temporarily assigning instances to disabled stop-words. After switching on the stop-word option again, words that are stop-words will be ignored.

Setting the stop-word switch on requires the definition of a stop-word collection for the Index Manager (configuration or ini-file). When no stop-word collection had been defined, the stop-word switch will be ignored.

### key\_word

When the option is switched on, the indexing process checks for defined keywords, only. All words not defined as keywords are ignored. This option is typically switched on for maintenance processes.

### auto\_key

When auto-key is on, the indexing process will add new words to the keyword collection without user interaction. This is a typical maintenance option and provides a fast way of building indexes (but with low quality).

### expand\_key

The expand-key option is able to handle multiple word keywords. Thus, it becomes possible to consider 'New York' as a single keyword.

Defining multiple word keywords is not subject of the Index Services. Your application must find an own way of defining multiple keywords.

A typical way is using defined concepts, which often consist of more than one word. When your application has good concept definitions, those can easily be copied to the keyword collection (DSC\_Concept → DSC\_Keyword).

Another way is importing multiple word keywords or adding those manually (e.g. in the Thesaurus application or via OShell).

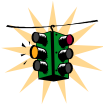
`remove_keys`

In order to remove old associations between object instances and keywords, this option should be switched on.

Indexing processes are, however, much faster, when this option is switched off. In this case, old keyword associations to the object instance are not removed and the object will still match old keywords.

`type_list`

The type list option must be switched on, when object type collections are to be created for each keyword. This will improve search performance and is required, when you search by type. Thus you may get separate search results e.g. for Persons and Cars referring to keyword 'blue'.



Do never change the option without switching on `remove_keys`, because type lists will be updated only for keywords associated with an object the first time.

## 8 Create application specific Index Services

Indexing logic becomes rather complex after a while and instead of running the Index Services 20 times or more, you may define you application specific index services.

### IndexManager class

Building index services is supported by the IndexManager class. Details for IndexManager class features are described in the reference documentation for this class.

The main targets of the class are

- Support indexing processes
- Support index search

### Opening IndexManager

To call IndexManager functions, the IndexManager must be opened first. Opening the IndexManager means creating an IndexManager objects and opening the keyword collections (keywords, stop-words and lexical base terms).

You may pass names for the keyword collections directly from within the program, but usually, you gain more flexibility, when passing collection names via the configuration or ini-file.

```
IndexManager im;  
if ( im.Open(db_handle,"Section1") ) ERROR
```

For opening the index manager, you may pass the name of a section defined in the configuration or ini-file, which has been passed to the function.

To prepare the next indexing step, you just need to call the Open() function once more, passing the section name for the specifications of the next step.

```
if ( im.Open(db_handle,"Section2") ) ERROR
```

Switches are set from the settings in the section of the configuration file. You may change settings for switches, since those are public in the IndexManager instance.

### Indexing process

During the indexing process, the index manager associates the objects from the object collection with existing keywords. Depending on the options set in the configuration file or by the program, the index processing stops at the next unknown keyword.

You may call the Run() function to run the default console processing. Since an indexing process works for a selected language, it might be necessary to set-up the language before running the indexing process.

```
Im.SetLanguage("English");
if ( im.Run() )                ERROR
Im.SetLanguage("German");
if ( im.Run() )                ERROR
```

If you want to provide your own handling for unknown keywords, you may write a simple loop as:

```
while ( im.Next() ) {
    if ( !(word = im.GetWord()) ) {
        // processing unknown word
    }
}
```

Processing unknown words you may call AddKeyword() or AddStopword() in order to create new keywords or stop-words. Before adding a keyword, you may provide a lexical base term.

```
while ( im.Next() ) {
    if ( !(word = im.GetWord()) ) {
        ...
        if ( IsKeyword(word) )                // application
function)
        im.SetLBTerm(GetLexicalBase(word));
        im.AddKeyword();
    }
}
```

In the example above, IsKeyword() and GetLexicalBase() are application functions providing algorithms for detecting keywords and assigning lexical base terms.

### Keyword search

The Index Manager supports keyword search by weighting objects relating to a keyword. Opening the index manager for keyword search requires a keyword collection, which might be defined in the ini-file or could be passed directly to the IndexManager constructor.

```
IndexManager im("DSC_Keyword");
if ( im.Open(db_handle) )                ERROR
```

Before calling Search() the application must provide a property handle, which will contain the result collection after searching. The result can be stored in a transient or temporary collection but also in a persistent collection in order to store the search result (optimizing search).

```
PropertyHandle    result;  
result.Open(GetDBHandle(), "KWSearchResult", PI_Write);
```

Here, the result collection had been defined as temporary extent in the application resource database.

After providing a result property handle, Search() can be called in order to obtain the result collection in the passed property handle.

```
// result and search_string          // passed as parameter  
if ( Search(db_handle, search_string, &result, NULL, 50) < 0 )  
    ERROR
```

Search returns the number of objects in the result collection. This is an estimated count, when the value is greater than the number of objects in the result collection. The number of objects in the result collection can be limited by the maximum count (50 in the example) passed to Search().

Maximum  
number

The number of objects in the result collection can be limited by the maximum count (50 in the example) passed to Search(). Limiting the result collection causes the Search() function to terminate, when the requested number of objects with the best rating had been found.

Since this is a rare case, usually Search works until the end. Thus, passing a maximum limit is rather a memory than a runtime optimization.

Type search

Search() supports searching for objects of a given type. Objects of different types associated with a keyword can be stored in type lists. When type lists had been created in the indexing processing, a type name can be passed to the search function in order to reduce the result to object instances of the passed type.

```
// result and search_string          // passed as parameter  
if ( Search(db_handle, search_string, &result, "DSC_Topic", 50) < 0  
)  
    ERROR
```

In this example, the search function will return topic objects (DSC\_Topic), only.

## Storing results

When searching frequently, it might be a good idea storing the result collections. Especially, when not using online re-indexing, the result sets will not change between two maintenance processes.

```
// result // passed as
parameter
NSString nsearch(search_string); // passed as
parameter
logical estimated = NO; // returned by search
int32 count = UNDEF;

if ( LocateKeywords(nsearch) ) ERROR
count = SearchByType(&result, "DSC_Topic", 50, estimated);
if ( count < 0 ) ERROR
```

LocateKeywords() returns a normalized search string, which allows identifying the search result. Using this string as key for storing the search result, instead of re-searching the result can be read directly from the database.

A timestamp in the search instance helps to keep stored results up-to-date and allows deciding when to re-evaluate the result.

## Extend key

When the keyword index supports extended keywords, i.e. keywords consisting of more than one word, the expand\_key option must be switched on in order to involve expanded key words. Expanded keywords will get higher weight than simple keywords.

The weight of an expanded keyword corresponds to the number of words it contains.

Keyword	Weight
new york city	6
new york	4

When a search string contains expanded key words, Search() looks for expanded keywords with a higher weight, that for simple keywords. Nevertheless, looking for "New York City" will result in five keyword entries used for searching when "new york city" and "new york" are stored as expanded keywords.

```
Search string: new york city traffic
Keywords      Weight
new york city 6
new york      4
new           1
york          1
city          1
traffic       2
```

Simple keywords in a search string will get the weight 2, while simple keywords extracted from an expanded keyword will get the weight 1.

#### Keyword delimiter

When passing a search string, keywords can be separated by any type of delimiter. Normalizing the search string will convert all delimiters into comma, except blanks between words in expanded keywords.

```
Search string:      new york city traffic
Normalizes string: new york city,traffic
```

Commas or other non-blank separators in the search string may, however, influence the result, since those are not allowed in expanded keywords.

```
Search string:      new york, city traffic
Normalizes string: new york,city,traffic
```

In the example above, “new york” is accepted as expanded keyword, only, since “city” had been separated by comma and does not count as part of an extended keyword.