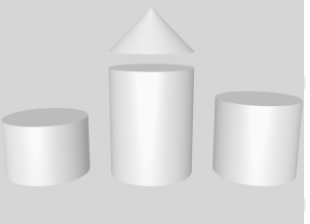


01101001001110010101  
10101101010010111011  
10001011101010101011  
10110010100101011010  
10101001101011010010  
01110010101101011010  
10010111011100010111



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

**Index Services**





**run Software-Werkstatt GmbH**  
**Weigandufer 45**  
**12059 Berlin**

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

Berlin, October 2012

# Content

- 1 Introduction .....4**
  - ODABA<sup>NG</sup> .....4
  - Platforms .....4
  - Interfaces.....4
  - User Interfaces .....4
  
- 2 Index Services .....5**
  - Tools.....5
  - Word collections .....5
  - Built index.....6
  - Multilingual support .....7
  - Index service application .....7
  - Index maintenance .....7
  
- 3 GUI Index Services .....8**
  - Ini-file .....8
  - Running GUI Index Services .....9
  - Predefined settings ..... 11
  - Indexing objects ..... 11
  
- 4 Console Index Services ..... 15**
  - Ini-file ..... 15
  - Maintenance options ..... 16
  - Running console Indexing Services ..... 17
  
- 5 Index Manager Options ..... 19**
  - collection ..... 19
  - field1...9 ..... 19
  - Switches ..... 20
  
- 6 Create application specific Index Services ..... 22**
  - IndexManager class ..... 22
  - Opening IndexManager..... 22
  - Indexing process ..... 22
  - Keyword search ..... 23

# 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 Index Services

Index Services are used for associating application object instances with a key word index. OODABA<sup>NG</sup> provides index services in order to generate keyword indexes for documentation elements (topics, concepts, themes).

In addition, Index Services support indexing any type of application object. The only requirement is, the application objects to be indexed inherit from the documentation object base (DSC\_Object). Since DSC\_Object is a system type, any application type may inherit from this type just by defining it as base type for the user-defined type.

Index Services support creating and maintaining indexes. In addition, ODABA<sup>NG</sup> provides an ObjectRating class, which produces search results.

Index Services allow indexing object instances, i.e. associating object instances with keywords. Object instances are associated with keywords found in selected text fields of the object or in text field of related objects.

Index Services support multilingual indexing, which allows creating indexes for different languages for multilingual text fields. Before scanning text fields, the text is normalized, i.e. HTML text fields are converted into plain text fields.

### Tools

For building indexes two tools are provided. The GUI Index Service provides a GUI tool, which supports building keyword collections and defining lexical base terms. The same can be achieved with the console Index Services.

#### GUI Index Services

GUI Index Services are provided to build keyword collections by indexing objects. This requires skills about indexing processes to provide high quality keyword collections.

#### Console Index Services

Console Index Services are mainly used to update keyword indexes without altering keyword collections. In order to run maintenance processes without user interaction, console applications can be set-up in full automatic manner.

### Word collections

Index creation is a rather difficult task, since during index creation, keyword and stop-word collections must be created. ODABA<sup>NG</sup> delivers a keyword and a stop-word

collection, but usually, those collections depend much on the subject area.

**Keyword collection** The keyword index contains all words used in object's text fields, which are considered as relevant. Usually, words like 'a' or 'the' would not count as relevant, since they are expected to appear in most text fields.

**Stop-word collection** In order to increase search efficiency, words considered as irrelevant are stored in a stop-word collection. Keyword collection and stop-word collection need not to be distinct. Thus, stop-word may act as keywords temporarily by switching off the stop-word feature.

**Lexical base** Different word forms may lead to strange results, since you might get different results when searching for 'property' or 'properties'. A lexical base is a word, which collects all word forms referring to the same concept or idea. Creating relationships between keywords and lexical base terms is a simple mean to improve the quality of an index.

**Keyword expansion** Expanded keywords are keywords consisting of more than one word. In the simple case, an object instance containing the text 'New York' would map to key words 'new' and 'york'. Searching for 'New York' in this case returns many uninteresting results. We would get much better results, when accepting 'New York' as one key word. Keyword expansion requires adding expanded keywords to the keyword collection manually.

**Built index** Building an index means creating the links between keywords and associated objects. After indexing database objects, each keyword refers to all object instances associated with that keyword.

Since you may index object instances of many different types, the object collection for a keyword is a weak-typed collection and may contain object instances of different types.

**Type lists** In order to improve comfort and performance for search requests, associated object instances can be ordered in type lists, where each list contains the objects for a certain type, only.

**Index creation with user interaction** Index creation with user interaction allows building high quality keyword and lexical base term collections. Hence, this technology should be used at the beginning of an indexing process. Later on, when keyword collec-

tions have been defined, automatic index creation is suggested for maintaining the index.

Automatic index creation

Indexes can be created automatically without any user interaction: When creating an index automatically, the indexing process may refer to a pre-defined set of keywords and stop-words. The indexing process may also create keywords automatically, in which case the created index must be cleaned up later on.

**Multilingual support**

Keywords, stop-words and lexical base terms support different languages. In order to create indexes for different languages, word collections can be translated. This provides an index that refers to object instances independent on the language used in the text fields of the objects.

In order to create different indexes for each language, keywords should not be translated but stored separately for each language. The current language for building an index is taken from the `DSC_Language` option, which can be set in the application or in the application.

**Index service application**

Most of the features required for indexing are provided in the `IndexManager` class. This allows writing simple applications for specific index processed instead of running the standard index service tools.

**Index maintenance**

Since text fields in object instances may change, keyword indexes must be maintained. This could be done in real time, i.e. when a text in an object had changed. This might become time consuming, especially when considering that only a few words had changed in a long text.

Another way is updating the keyword-object relationship in a regular maintenance process. Since in this process many objects have to be checked, but just a few had changed, the index can be updated by including objects only, which had changed the last indexing process.

### 3 GUI Index Services

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

ODABA/ode90.exe *ini-file*

#### Ini-file

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

#### **[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=!\opa\opa.dev

NET=YES

ONLINE\_VERSION=YES

ACCESS\_MODE=Write

#### **[IndexManager]**

keywords=DSC\_Keyword

stopwords=DSC\_Stopword

lexterms=DSC\_LexTerm

#### [SYSTEM]

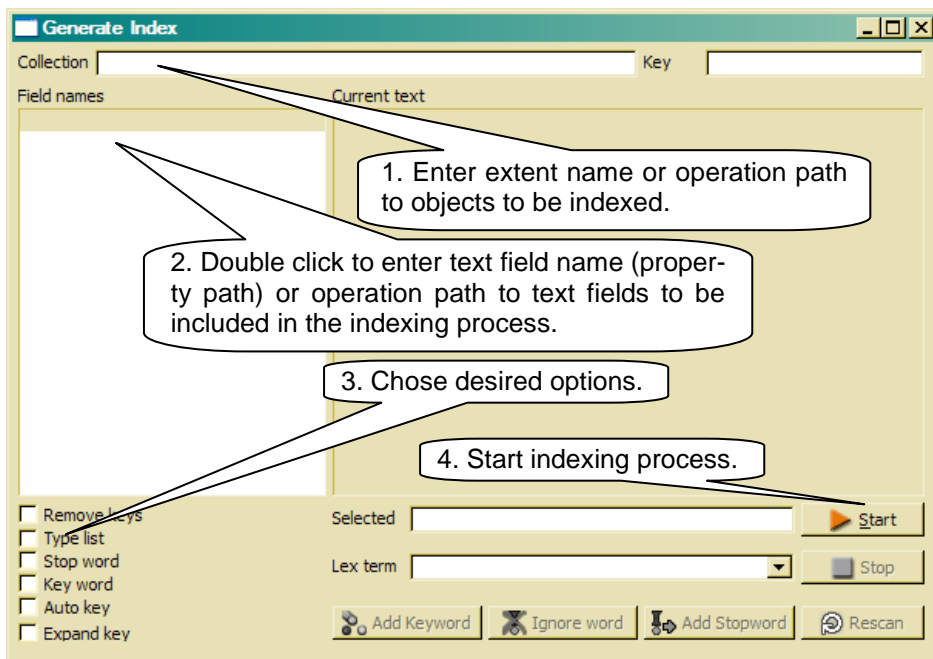
The system section refers to database system information. The minimum required is the DICTIONARY reference to the system dictionary, which is stored in the ODABA<sup>NG</sup> installation folder. When running the application with a system dictionary stored on the server, server name and a port number 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

## Predefined settings

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

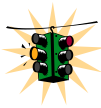
```
keywords=DSC_Keyword  
stopwords=DSC_Stopword  
lexterms=DSC_LexTerm
```

```
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
```

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.



collection

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.

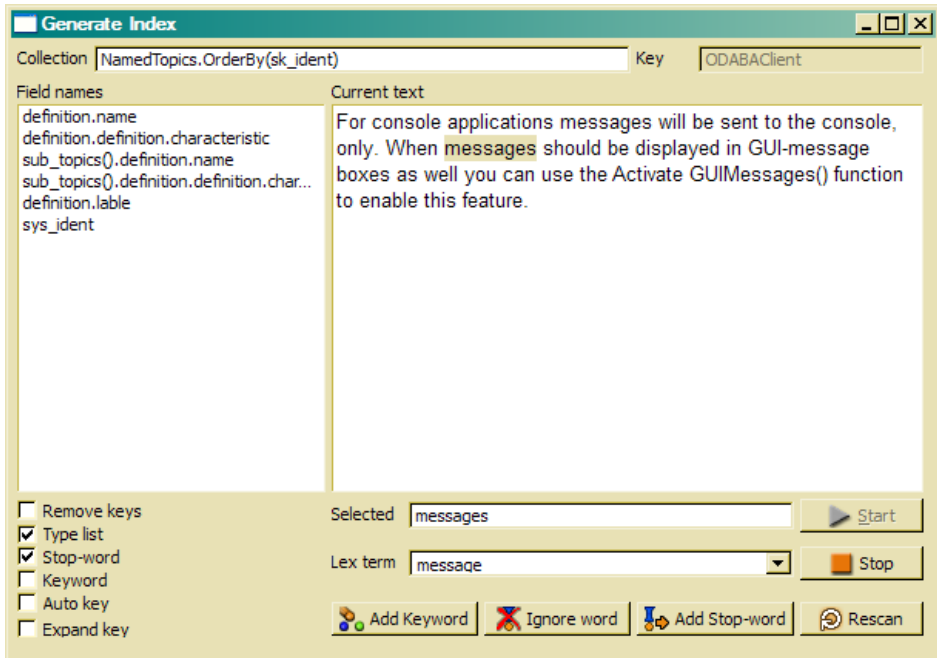
Field 3 and 4 refer to a collection of subordinated text fields, which are included into the evaluation as well.

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 it 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 <b>Ignore word</b> .
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 <b>Rescan</b> 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 <b>Keyword</b> 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 <b>Key</b> field above the text field.



## 4 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=!:lopa\lopa.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 ODABA<sup>NG</sup> 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.

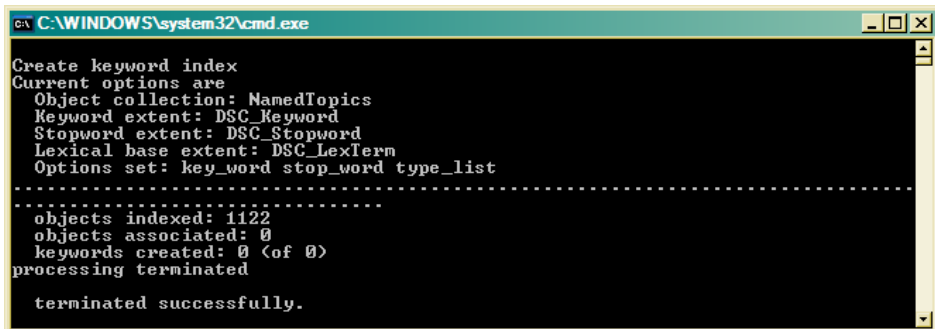
Optimal solution The optimal solution would be to run a sort of auto-matching between new keywords and lexical base terms. Still, we will need a good tool for maintaining mismatches and turning keywords into stop-words.

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.

## 5 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.

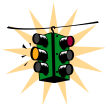
```
collection=NamedTopics().sub_topics
```

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).

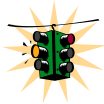
remove\_keys

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.

## 6 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 GetLexical-Base() 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.