

---

# XMLmind Spell Checker - SDK Tutorial

Xavier Franc, Pixware <xsc-support@xmlmind.com>

## Abstract

This tutorial explains how to add spell-checking functionalities to your Java applications, using the XMLmind Spell Checker SDK.

It assumes that the reader has some familiarity with Java™ programming and Swing™.

1. Using the evaluation version of the SDK .....	1
2. Introduction .....	1
3. User Interface .....	2
3.1. Simplest Integration .....	2
3.2. Automatic spell-checking during edition .....	2
3.3. Manipulation of JSpellDialog and its components .....	3
3.4. Adapting to other text sources .....	3
3.5. Text Parsing Helpers .....	3
4. Core Engine .....	4
4.1. Services provided by SpellChecker .....	4
4.2. Step-by-step .....	4
4.3. Options .....	7
4.4. Manipulating Dictionaries .....	8
5. Using the Spell Checker with Java Web Start .....	9

## 1. Using the evaluation version of the SDK

*If you have purchased XMLmind Spell Checker, please skip this section.*

Downloading and installing the evaluation version of the SDK is not sufficient to use the spell checker, you must also get an evaluation key and install this key in the product. In order to do that, please proceed as follows:

1. Please fill this form [http://www.xmlmind.com:8080/eval\\_key/generate\\_key](http://www.xmlmind.com:8080/eval_key/generate_key) in order to be sent an evaluation key. This evaluation key is contained in the attachment of the email message you'll receive. This key is packaged in a file called `xsc_eval_key.jar`.
2. Copy `xsc_eval_key.jar` next to `xsc.jar`.
3. This is sufficient if you just intend to use one of our sample programs (e.g. `xSpell`).

However, if you integrate the spell checker engine in your own programs, do not forget to also reference `xsc_eval_key.jar` wherever you reference `xsc.jar`. Generally, this means adding both `xsc.jar` and `xsc_eval_key.jar` to your `CLASSPATH`.

Please note that after 30 days of evaluation, the spell checker engine will throw a `RuntimeException` each time you'll attempt to use it.

## 2. Introduction

The SDK components can be utilized at two levels:

1. *User Interface*: the classes in the package `com.xmlmind.spellcheck.ui` form a Swing User Interface built above the core Spell Checker engine. This package is the one to be used in most applications.

A very easy integration can be achieved by using a few static methods in the class `JSpellDialog`.

2. *Core Engine*: meant for advanced usage, or for batch-style applications where no User Interface is needed. The package is `com.xmlmind.spellcheck.engine`.

## Note

In the documentation and in the API, the word "language" is often used in place of "dictionary": this is because there is generally one dictionary per language, and the short language name (for example "en") can be used instead of the longer name "en/default". But it is possible for a language to have several different dictionaries.

## 3. User Interface

```
[package com.xmlmind.spellcheck.ui]
```

This section explains how to use the high-level User Interface components.

We start by explaining the simplest way of invoking the Spell Checker on Swing text components (one line of code), then we gradually go into more advanced topics.

### 3.1. Simplest Integration

In order to popup the Spell Checker dialog in your Swing Text Component, you need only to import the `com.xmlmind.spellcheck.ui.JSpellDialog` class, then place the following code in the implementation of your "spell-check" command (for example a menu item, or a tool-bar button):

```
JSpellDialog.check( textComponent, language )
```

where `textComponent` is a Swing Text component (an object inheriting `JTextComponent`), and `language` is the short name of a dictionary or language (in the standard distribution: "en", "fr", "de" or a country variant "en-US", "fr-CA", etc). It can be null (the previous dictionary is then used). By default, "en" is used.

Note that this assumes a standard installation of the dictionaries (typically in a directory `dicts/` located at the same place as the SDK's jar, `xsc.jar`, or in a *Dictionary Archive* `.dar` similar to a Jar file and placed in the `CLASSPATH`).

### 3.2. Automatic spell-checking during edition

Also referred to in literature as "Continuous", "On the fly", "Real Time", etc, spell-checking, this feature marks erroneous words with red underlining during normal edition, without the need to invoke a Spell Checker dialog.

It is very easily activated (typically just after the document is loaded into the text component):

```
JSpellDialog.enableAutoCheck( textComponent, language )
```

The `textComponent` and `language` parameters have the same meaning as before.

It is safe to invoke this method twice on the same component (for example to change the dictionary).

To deactivate the feature (typically when the document is closed), call:

```
JSpellDialog.disableAutoCheck( myTextComponent )
```

When an erroneous word is underlined, the user expects to be able to right-click on it to display a popup menu which will allow her/him to fix the spelling error. In order to achieve that, include the following code snippet in your application.

```
textComponent.addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent e) {  
        showAutoCheckMenu(e);  
    }  
})
```

```
public void mouseReleased(MouseEvent e) {
    showAutoCheckMenu(e);
}

private void showAutoCheckMenu(MouseEvent e) {
    if (e.isPopupTrigger())
        JSpellDialog.showAutoCheckMenu(textComponent,
                                        e.getX(), e.getY());
}
});
```

The following API (which returns a `boolean`) can be used to test if it makes sense to display the spell-checking popup menu in the current context and for the specified location.

```
JSpellDialog.canShowAutoCheckMenu( textComponent, mouseClickX, mouseClickY )
```

This API is useful

- if you want display the spell-checking popup menu when the user clicks on an underlined, erroneous, word;
- if you want to display your regular popup menu otherwise.

### 3.3. Manipulation of JSpellDialog and its components

The static convenience methods described above use a static `JSpellDialog` object.

If this is a problem, it is possible to instantiate one's own `JSpellDialog`, for example like this:

```
JSpellDialog dialog =
    new JSpellDialog( myFrame, myTextComponent, null, language );
// ...
dialog.popup(true);
```

For fine control of the Spell Checker, it is generally necessary to get the `JSpellComponent` contained in the `JSpellDialog`, and then use its methods.

`JSpellComponent` implements the actual spell-checking interaction, it is a component embeddable in any Swing user interface. `JSpellDialog` simply embeds it in a Swing dialog. To know more about `JSpellComponent`, please see its Java documentation.

`JSpellComponent` has an associated option dialog (`JSpellOptions`).

The XSpell sample application (source code found in `samples/xspell/src/XSpell.java`) shows an example of use of the GUI package. It also defines and uses different text parsers to create styled documents from common text formats (Javadoc comments, HTML, etc)

### 3.4. Adapting to other text sources

To make the Spell Checker work on text sources different than `JTextComponent`, the recommended way is to create an adapter that implements the interface `com.xmlmind.spellcheck.ui.TextSource`. Then `JSpellDialog` can be used with this adapter.

If one is not satisfied with this scheme, it is necessary to work directly with the Core Engine (see following section).

### 3.5. Text Parsing Helpers

There are additional classes in `com.xmlmind.spellcheck.ui` that provide a framework and sample code for parsing miscellaneous text formats: `TextParser` is an abstract class, and `MLParser` an implementation that can read XML/HTML/SGML or other XML-like markup formats. An example of utilization of these classes is provided in `samples/mlscan/MLScan.java`.

## 4. Core Engine

[package com.xmlmind.spellcheck.engine]

Here, there is no notion of graphical user interface. This section is meant for developers who want to modify the GUI components, or create entirely different applications. This can be the case, for example, if you want to do batch processing or modify the servlets of the Client/Server edition.

- The central class is `SpellChecker`. It is a "facade" that provides most of the services.
- The `Suggestions` interface gives access to the suggestions produced by `SpellChecker`.
- Access to text is abstracted by a very simple interface, `CharSequence`. It is analogous to the `java.lang.CharSequence` interface of JRE 1.4 and in the future will be deprecated in favor of the standard.

A few simple implementations of this interface are available in package `com.xmlmind.spellcheck.util`.

- There is another class, `DictionaryManager`, which needs not be manipulated by simple applications. This class has to be handled in the case you want to share dictionaries among several instances of `SpellChecker` (for example in a server-side application), or if you want to access additional dictionaries in non-standard locations.

### 4.1. Services provided by SpellChecker

They come in different categories:

- Checking work per se: scanning a sequence of characters, checking individual words, getting suggestions for an erroneous word,
- Manipulating languages and dictionaries, in particular the personal dictionary (learning words and suggestions).
- Setting and getting miscellaneous control options.

#### About multi-thread safety

The `SpellChecker` class cannot be shared between different threads since it contains the state of a spelling session.

Therefore in the context of a multi-threaded server for example, one instance of `SpellChecker` must be created for each spelling session. Since `SpellChecker` is not a heavyweight object (it does not contain the compiled dictionaries), there is no performance issue.

At the opposite `DictionaryManager` (used by `SpellChecker`) is multi-threadable and its purpose is precisely to share dictionaries. See the section about dictionaries for more details.

### 4.2. Step-by-step

1. The first step is to create an instance of `SpellChecker`.

This is performed directly. The simplest constructor has one argument, which is a "dictpath", the name of a directory where dictionaries are installed.

For a discussion of the dictionary storage conventions, see the "Dictionary Management" section below.

```
String dictPath = ... // application-dependent
SpellChecker checker = new SpellChecker( dictPath );
```

2. Then it is possible to set a number of options: see the 'Options' section for more details.

It is important to set the current language: `setSelectedLanguage(languageCode)` performs this task. Available languages can be obtained by `listLanguages()`.

Something useful is the load and save path for the personal dictionaries (There is a distinct personal dictionary for each language). This is probably application and system-dependent.

```
String path = homeDir + File.separatorChar
              + "myapp_spell" + File.separatorChar + "%L%" ;
checker.setPersonalDictionaryPath( path );
```

Actually the path set is a pattern that has to contain a marker for the language name. The marker for the language name is "%L%", as it can be seen in the example above.

3. Then we come to the spell-checking main loop. The model we use is very simple and attempts to make as few hypotheses as possible about the client application.

The actual implementation of your text is abstracted by an interface called `CharSequence` (the same as in the JRE 1.4).

The Spell Checker accepts a piece of text so a `CharSequence` through `SpellChecker.setInput(...)`, then `checkNext()` is invoked.

```
CharSequence myInput = ... // get from your application
checker.setInput( myInput );
int err = checker.checkNext();
```

If `checkNext()` returns `ERR_NONE`, the piece of text set as input is correct, and the application has to proceed on the next piece of text or to finish.

4. Processing the errors returned by `checkNext()`:

#### `ERR_NONE`

No error has been detected. The application should proceed to the next piece of text, or to finish.

#### `ERR_UNKNOWN_WORD`

A word not contained in dictionaries, and not compound from existing words. Typically, you will invoke `getSuggestions()` to obtain pertinent (we hope so) suggestions for correcting the word.

#### `ERR_WRONG_CAP`

The word is known, but is improperly capitalized. For example it is a proper name starting with a lowercase letter, or an acronym expected to be in all caps (for example "Xml" instead of "XML"). It can also be a plain word after an end-of-sentence punctuation mark. It is also possible here to invoke `getSuggestions()` which should return the properly capitalized word first among other suggestions.

This error can be inhibited by setting the `CheckCase` option to false.

#### `ERR_PUNCTUATION`

A dubious sequence of punctuation marks was found: either a whitespace before marks such as dot, comma, colon, semicolon, question or exclamation mark, or two consecutive marks (except dots) such as ".,". Here also, `getSuggestions()` will propose replacements.

This error can be inhibited by setting the `CheckPunctuation` option to false.

#### `ERR_DUPLICATE`

Two identical consecutive words. (Note: In some languages it can sometimes be correct, like in English "had had" or in French "nous nous", but this case is not yet supported). `getWord()` and `getPosition()` return the second word, but `getSuggestions()` does not return proper results. The action is basically to ignore the error or to delete the second word.

This error can be inhibited by setting the `CheckDuplicate` option to false.

**ERR\_REPLACE**

If the personal dictionary has been enriched with replacements to perform automatically (using `learnAutoReplacement()`) - this corresponds with a command like "Replace Always"- the `checkNext()` method signals it has encountered such a replacement. The action to take here is to invoke `getReplacement()` passing the word obtained by `getWord()`, the to proceed in the check loop. For example:

```
String word = checker.getWord();
myTextSource.replace( checker.getPosition(), word.length(),
                    checker.getReplacement(word) );
```

This mechanism can be inhibited by setting the `AutoReplace` option to false.

**5. Notes about `setInput()` and the checking loop:**

The character sequence set with `setInput()` is assumed to stay unmodified by the application until `checkNext()` reaches its end. Depending on the implementation, this can often be unrealistic if a replacement is performed. Therefore:

- the simplest way is to always call the `setInput()` method before `checkNext()`, with a fragment reflecting the updated state of the text source.
- Alternately, the input text can be left untouched by the modifications, but the application has to translate the positions returned by `getPosition()`, since they are relative to the original text fragment

**6. A skeleton of the search loop:**

Note: The text source (here `mySource`) typically implements the `TextSource` interface defined in package `com.xmlmind.spellcheck.ui`.

```
void doSearch() {
    for(;;)
    {
        ... // prepare
        // acquire next fragment from application:
        input = mySource.getText(checker.getCharChecker());
        if (input == null) {
            ... // no more input
            return;
        }
        checker.setInput(input);
        int err = checker.checkNext();
        if (err == SpellChecker.ERR_NONE) {
            // end reached: update position in source
            ...
            continue;
        }
        String failingWord = checker.getWord();
        int replacePos = checker.getPosition();
        int replaceSize = failingWord.length();
        // application dependent:
        mySource.highlight(replacePos, replaceSize);

        switch(err) {
            case SpellChecker.ERR_DUPLICATE:
                showStatus("duplicate word: " + failingWord);
                break;
            case SpellChecker.ERR_REPLACE:
                mySource.replace(replacePos, replaceSize,
                               checker.getReplacement(failingWord));
                continue;
            case SpellChecker.ERR_WRONG_CAP:
                showSuggestions("word should be capitalized");
                break;
            case SpellChecker.ERR_PUNCTUATION:
                showSuggestions("punctuation problem");
                break;
        }
    }
}
```

```
        case SpellChecker.ERR_UNKNOWN_WORD:
            showSuggestions("unrecognized word");
            break;
        }
        // get and process user commands:
        break;
    }
}
```

### 7. *Displaying suggestions:*

The interface `Suggestions` returned by `SpellChecker.getSuggestions()` provides methods to retrieve suggestions individually: `getSuggestion(int index)` or as an array: `String[] Suggestions.toArray()`.

Suggestions are ordered by decreasing pertinence and their number is given by `Suggestions.getCount()`.

The maximum number of returned suggestions can be set by `setSuggestionLimit`.

Example:

```
String[] suggestions = checker.getSuggestions().toArray();
JList displayList = new JList(suggestions);
if (suggestions.length > 0)
    displayList.setSelectedIndex(0);
```

### 8. *Smarter suggestions:*

There is a mechanism to teach the `SpellChecker` to make better suggestions. When a word entered by a user to correct an erroneous word is not among the first suggestions found, it is possible to invoke `learnSuggestion()` with the wrong word and its correction as arguments: the next time this word is encountered, the learned suggestion will be put atop the suggestion list.

```
void doReplace() {
    String correction = ...; // get correction from user
    // if not among the 3 first , learn it:
    if (!suggestions.contains( correction, 3 ))
        checker.learnSuggestion( failingWord, correction,
                                SpellChecker.TEMPORARY_DICT );
    ...
}
```

In this example, the learned suggestion is put into the temporary dictionary, therefore lost at the end of the session. It is also possible to put it in the persistent personal dictionary (`SpellChecker.PERSONAL_DICT`).

## 4.3. Options

Options are manipulated in a get/set way (to be compatible with the Java Bean requirements).

For example, the `IgnoreCase` option is handled with `boolean getIgnoreCase()` and `void setIgnoreCase(boolean)`.

`SpellChecker` has also two methods (`loadOptions` and `saveOptions`) to globally set/retrieve options from/into a `java.util.Properties` object.

**Table 1. Options**

Option	Description	Type	Default value
IgnoreCase	if set, ignore capitalization errors	boolean	false
IgnoreMixed-Case	If set, do not check words containing case mixing (e.g. "SpellChecker")	boolean	false
IgnoreDigits	If set, do not check words containing digits (e.g. "b2b")	boolean	true
IgnoreURL	If set, ignore words looking like URL or file names (e.g. "www.xxx.com" or "c:\boot.ini")	boolean	true
IgnoreDuplicates	If set, do not signal two successive identical words as an error.	boolean	false
CheckPunctuation	If set, punctuation checking is enabled: misplaced white space and wrong sequences, like a dot following a comma, are detected.	boolean	false
AllowCompound	If set, all words formed by concatenating two legal words with an hyphen are accepted. If the language allows it, two words concatenated without hyphen are also accepted.	boolean	true
AllowPrefixes	If set, a word formed by concatenating a registered prefix and a legal word is accepted. For example if "mini-" is a registered prefix, accepts "mini-computer".	boolean	true
AllowFileExt	If set, accepts any word ending with registered file extensions (e.g. "myfile.txt", "index.html" etc.)	boolean	true
AutoReplace	Enables the "Replace Always" feature. If set, the <code>checkNext</code> method of <code>SpellChecker</code> can return <code>ERR_REPLACE</code> , then <code>getReplacement()</code> can be used to retrieve the replacement value.	boolean	true
SuggestionForce	Intensity of suggestion search: ranges from 0 to <code>FORCE_MAX</code> .	int	<code>FORCE_DEFAULT</code>
SuggestionLimit	Maximum number of suggestions returned (does not influence the duration of a suggestion search).	int	15

## 4.4. Manipulating Dictionaries

There are numerous methods for managing dictionaries.

To know more about dictionary structure, read the Dictionary Builder documentation.

The most likely used methods are the following:

- `setPersonalDictionaryPath`: defines a pattern for file storage location for personal dictionaries.
- `listLanguages`: returns a list of items described detected languages and dictionaries.
- `setSelectedLanguage`: selects a language, loads default dictionary if necessary.
- `selectDictionary`: loads a dictionary (if necessary) and selects implicitly the dictionary's language. This method works like `setSelectedLanguage`, except that other dictionaries already loaded in the same language are removed.
- `getSelectedLanguage` `getSelectedLanguageInfo`: information about the currently selected language.
- `savePersonalDictionaries`: forces a save of all personal dictionaries (for example on exit).
- `getDictionaryManager` `setDictionaryManager`: for more advanced control.

- `setDictionaryPath`: defines a non-standard directory where dictionary archives (`.dar`) can be found.

Other methods:

- `clearLanguageDictionaries`: resets a language.
- `listEditableDictionaries`: returns a list of editable dictionaries for the current language.
- `manageEditableDictionary`: to select, add, load, or remove an editable dictionary.
- `getEditableWords`: returns an array of word descriptors from the current editable dictionary.
- `changeWord`: to edit the contents of editable dictionaries.

## 5. Using the Spell Checker with Java Web Start

XMLmind Spell Checker supports the Java Web Start technology. This technology simplifies the deployment and maintenance of Java applications. See the Sun product page for more information.

For application designers, the basic constraint of Web Start is that all resources for the application must reside in jar files, and accessed through the `ClassLoader.getResource()` method. Unfortunately there is no way to list available resources and this was making dictionary management difficult in earlier versions of XMLmind Spell Checker.

To use XMLmind Spell Checker in a Web Start-enabled application, there is no other requirement than to provide the dictionaries in a specially prepared DAR archive. This archive (which is in fact a `jar`) will be a component of the Web Start application, as well as `xsc.jar`.

For example, here is the resources section of XSpell's JNLP descriptor<sup>1</sup>:

```
<resources>
  <j2se version="1.4+" />
  <jar href="XSpell.jar" main="true" download="eager" />
  <jar href="xsc.jar" download="eager" />
  <jar href="dicts.dar" download="eager" />
</resources>
```

A ready to use `dicts.dar` is included in the SDK. It contains all the dictionaries provided by XMLmind. To modify this archive, for example add a dictionary to it, see the Dictionary Builder documentation.

---

<sup>1</sup>When using the evaluation version of the SDK, this resources section becomes:

```
<resources>
  <j2se version="1.4+" />
  <jar href="XSpell.jar" main="true" download="eager" />
  <jar href="xsc.jar" download="eager" />
  <jar href="xsc_eval_key.jar" download="eager" />
  <jar href="dicts.dar" download="eager" />
</resources>
```