

Multilingual

March 10, 2015

Overview

The Multilingual set of languages supports creating MPS content in several human languages. As the term *language* is already highly overloaded in MPS, we stick to the term *multilingual* (sometimes also called **Internationalization**, **i18n**, or **Translation**).

Fundamentally, we stick to the common way of translating Java applications, as described in the Java documentation at <http://docs.oracle.com/javase/tutorial/i18n/>. This means every multilingual string refers to a `messageKey`. The system tries to resolve this key to an entry in a Java Properties file assigned to the the current (human) language. These strings can be part of regular BaseLanguage code or constants in MPS editors.

We also support translated domain elements (aka Concepts). Think of a Todo-List implemented as an MPS language, and every task on the list can be entered and displayed in several (human) languages.

Terms

Multilingual	All of the following MPS languages: <code>com.mbeddr.mpstutil.multilingual.common</code> , <code>com.mbeddr.mpsutil.multilingual.common</code>
MessageKey	Instance of <code>MessageKey</code> representing one message shown to the user.
ResourceBundle	Instance of <code>ResourceBundle</code> containing all messages used in the same model.
Current Language	The human language currently selected to be shown to the user.

Settings

Multilingual adds its own Project Settings page to the MPS Settings dialog (Fig.1). It's on Project scope because the settings might be shared in a team.



Figure 1: Multilingual Settings Dialog

The available settings are:

- Show Translations Globally: "Master switch" to enable display of multilingual elements at all.

- Enable "Show Translations" Intention: If the intention (see Chapter) should be available.
- Current Language: Select the `currentLanguage`. By default, this language is the current `Locale`.

Key Lookup

For both Chapter and Chapter, the actually used string is looked up by the referenced `messageKey`. This happens only at runtime, i. e. when the `BaseLanguage` code is executed or the editor is displayed. At creation time, the *default* value of the `MessageKey` is displayed all the time.

The *default* value will also be used if the key cannot be resolved by the `resourceBundle`.

A `ResourceBundle` calls `getBundle` to get the appropriate Java `ResourceBundle`. The parameters are:

- `baseName`: The value of property `baseName`.
- `locale`: The locale of the `currentLanguage`.
- `classLoader`: The `ClassLoader` of the Module the `ResourceBundle` is contained in.

The method `getString` is called with the used key as parameter. The key is composed of:

- The value of property `keyPrefix`, if not empty.
- The value of property `technicalKey`, if not empty.
- The value of property `MessageKey.name`, if no `technicalKey` was given.

If we enter a `MessageKey` that does not exist yet, there's an intention to create this key in the next `ResourceBundle` nearby.

Using Multilingual in `BaseLanguage`

We support two kinds of multilingual strings:

- `MultilingualJavaString` for simple Java strings.
- `MultilingualJavaRichString` for formatted Java strings, as defined in `format`.

Both can be used at any place a simple Java string constant can be used. Make sure the MPS language `com.mbeddr.mpsutil.multilingual.baseLanguage` is listed as used language.

The type of both of these strings is `MultilingualJavaStringType`, which is a subtype of `string`.

In order to enter a multilingual string, and typing with an `@` (at symbol); it will be shown as a flag icon once we selected the desired kind of multilingual string.

For a simple string, continue with `"` (double quotation mark).

The more elaborate formatted version continues with `'`' (three single apostrophs).

Note that we can use all format specifiers available in Java.

```

public string simpleString() {
    string myString = "hello";
    multilingual string myTranslatedString = 🇸🇪"this will be translated";

    string myComposedString = myString + " " + myTranslatedString + 🇸🇪"is cool!";

    return myComposedString;
}

```

Figure 2: Example of MultilingualJavaString usage.

```

public string formattedString() {
    string myString = "hello";
    int myInt = 42;
    double myDouble = 3.141592;

    multilingual string myTranslatedString =
        🇸🇪formattedKey "'This contains some                                ... {myString, myDouble, myInt };
        %1$:                                [general[Object]]
        conditional cell factory not installed
        string and a fancy
        %2$020.50:                                [floating-point[double]]
        conditional cell factory not installed
        float. Another thing is an
        %3$:                                [integral[Long]]
        conditional cell factory not installed
        Long

    return "RETURNED " + myTranslatedString;
}

```

Figure 3: Example of MultilingualJavaRichString usage.

The parameters required by a RichStringMessageKey are passed to the MultilingualJavaRichString. They are checked for matching the correct type.

The actual default values (also for MultilingualJavaRichString) can only be edited in the resourceBundle For reference, the ResourceBundle used for the examples of this section are shown in Fig.4.

Multilingual Resource Bundle DocsResourceBundle

baseName <no baseName>

key prefix <no keyPrefix>

keys

Key	Default	Technical Key
thisWillBeTranslated	<i>this will be translated</i>	
isCool	is cool!	
formattedKey	<pre> '''This contains some ... %1\$: [general[Object]] conditional cell factory not installed string and a fancy %2\$020.50: [floating-point[double]] conditional cell factory not installed float. Another thing is an %3\$: [integral[Long]] conditional cell factory not installed Long </pre>	

Figure 4: ResourceBundle used for examples in this section.

Using Multilingual in Concept Editors

We can translate two different things in a Concept Editor:

- MultilingualConstant translates constant strings in a concept editor.
- MultilingualAlias translates the alias of a concept.

```

<default> editor for concept DocMultilingualConcept
node cell layout:
  [> 🇵🇹 #alias# { id } 🇵🇹 This is some translated constant. <]

inspected cell layout:
  <choose cell model>

```

Figure 5: Example usages of MultilingualConstant and MultilingualAlias in Concept Editor

MultilingualConstant looks up the `messageKey` as described in Chapter. MultilingualAlias uses the `alias` (without any prefixes) as key for the `ResourceBundle`. The `resourceBundle` should be contained in the editor aspect containing the MultilingualConstant or MultilingualAlias.

Using Multilingual in Domain Elements (aka Concepts)
 For both multilingual in BaseLanguage and multilingual in Concept Editors, the translation is provided as part of development. In contrast, multilingual in Domain Elements is meant to create MPS Language Concepts hosting content in different human languages. Fig.6 shows the structure of multilingual concepts.

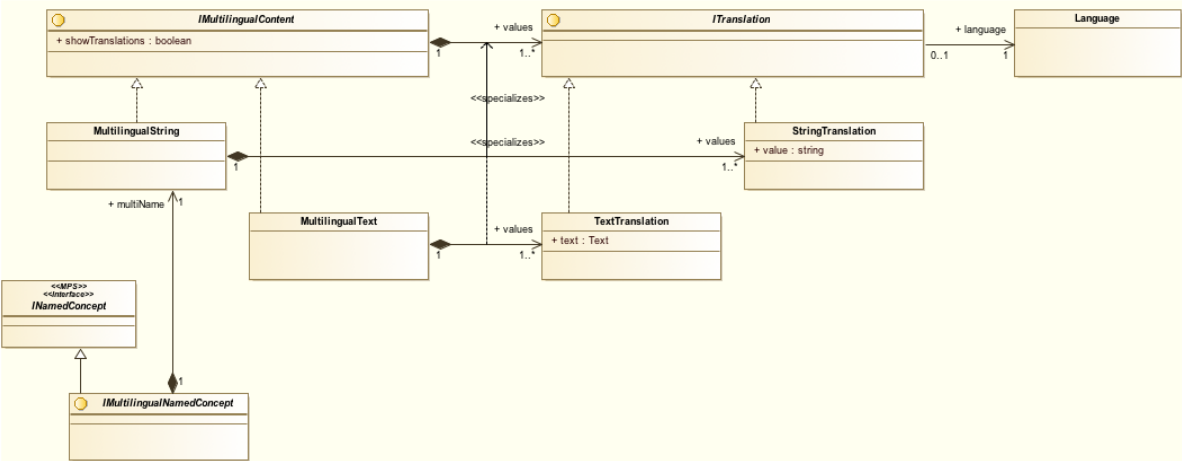


Figure 6: Concept structure for Multilingual Concepts

From a developer perspective, we're mostly interested in `IMultilingualContent` and its subconcepts `MultilingualString` and `MultilingualText`. They are meant as drop-in replacements for a simple `string` property (in case of `MultilingualString`) or `Text` (in case of `MultilingualText`).

Internally, they contain one or more `StringTranslation` or `TextTranslation`, respectively. Each `ITranslation` hosts a translation into one `Language`.

By default, the user can enter strings or texts into the multilingual concepts just as if they were their plain counterparts. They are considered the translation into the `currentLanguage` (Fig.7).


If the setting `settingShowTranslationsGlobally` is `true`, a flag symbol is displayed at each instance of `IMultilingualContent` (Fig.8).

Example Concept Very new Example Node

Description:

This is an even more fancy description.

Figure 7: Multilingual Concept without indication.

Example Concept Very new Example Node 


Description:

This is an even more fancy description.



Figure 8: Multilingual Concept showing only the current language.

If the property `showTranslations` is set to `true`, a table of all available `ITranslations` for this node is displayed. The user can add, edit, or remove any of them. This property can be toggled by an intention if the setting `settingEnableIntention` is set to `true` (Fig.9).

Example Concept English (United States) Very new Example Node 
Deutsch (Deutschland) Sehr neuer Beispielknoten

Description:


English (United States)	This is an even more fancy description.	
Deutsch (Deutschland)	Das ist eine noch ausgefallenerere Beschreibung.	
français	Ce est une description encore plus de fantaisie.	

Figure 9: Multilingual Concept showing all available translations.

If an `IMultilingualContent` is queried for its content, it returns the `ITranslation` for the `currentLanguage`. If this translation is not available, it returns the first translation.

For convenience, we also provide the concept interface `IMultilingualNamedConcept`. Its meant as a drop-in replacement for `INamedConcept`, replacing the `name` property by its multilingual counterpart.

Providing translated resources

As per Java convention, the translated resources follow the rules described in <http://docs.oracle.com/javase/7/docs/api/java/util/ResourceBundle.html#getBundle%28java.lang.String,%20java.util.Locale,%20java.lang.ClassLoader%29>.

Currently, MPS reliably supports loading resources only from jar files. Therefore, we're advised to package our properties files into jar archives.

Make sure to match the `baseName` of your resource bundle and the `messageKey` of your key, as described in Chapter. Add the jar to both Module Properties

- Common Tab, Java Classes Model Root (Fig.10)
- Java Tab, Libraries (Fig.11)

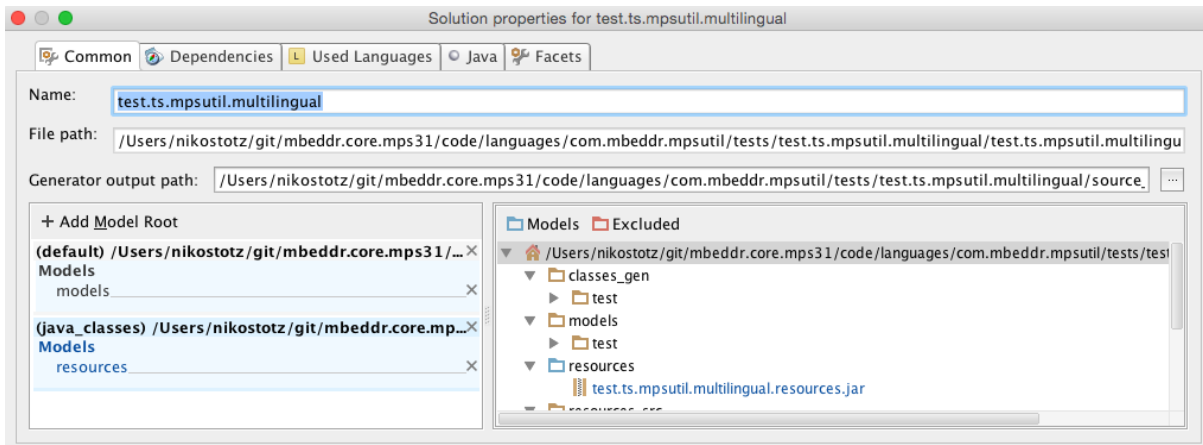


Figure 10: Example of resources jar in Module Common Tab.

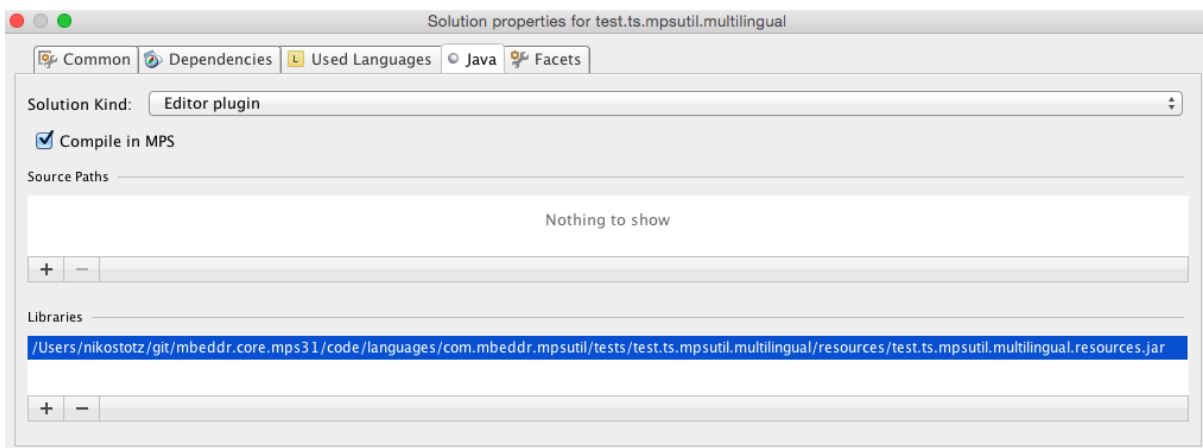


Figure 11: Example of resources jar in Module Java Tab.

In the directory structure example shown in Fig.12, we use the baseName `test.ts.mpsutil.multilingu`

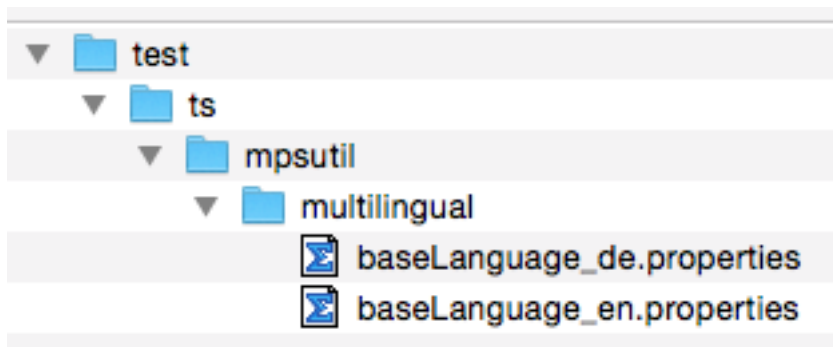


Figure 12: Directory structure to be used for ResourceBundle baseName `test.ts.mpsutil.multilingual.baseLanguage`.