JMLS

JMLS is a small java multi language supporter for graphical user interfaces. To use JMLS all GUI components will be generated by JMLS. The language specific parameters like icons, tooltips and labels will be defined in a language file for each language. The language file offers some more possibilies than the `Java ResourceBundle` and makes it easier to support more languages.

Some Features

- includes nearly all java swing components
- easy to add icons to the components
- supports some custom components
- supports frames and tabbed panes
- automatic generation of a language menu
- works for java applications and JApplets

Building and Requirements

In order to build a *.jar file JMLS requires:

- Maven - is used to resolve dependencies.
- JDK 7+ - JMLS is developed with Oracle JDK 7 but should also work with OpenJDK.

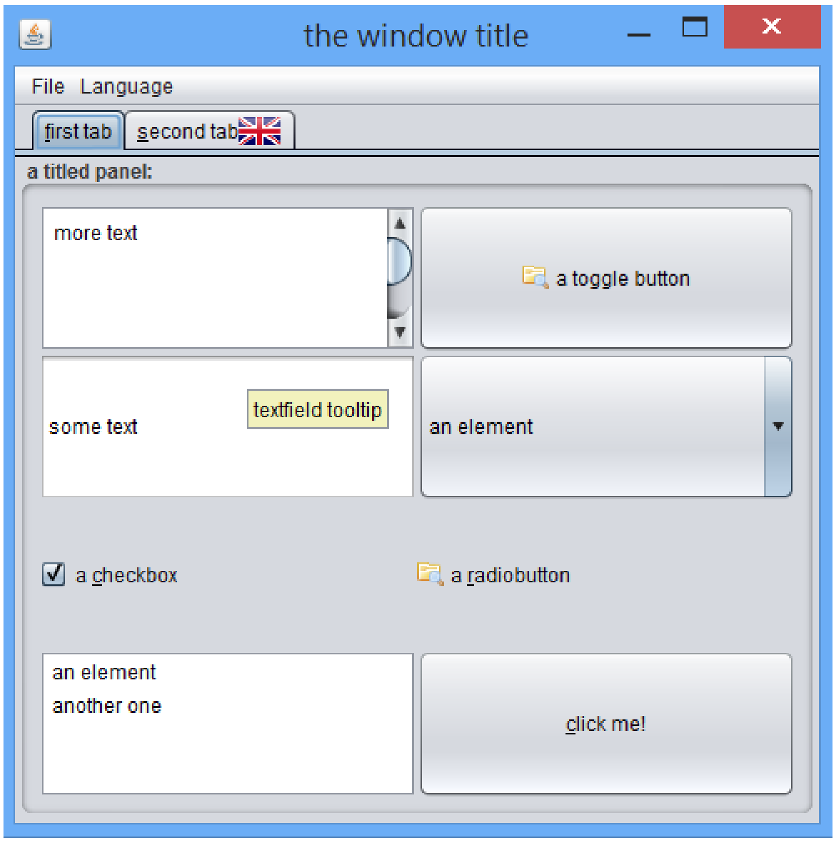
To build JMLS use the following command:

```
mvn clean package
```

The *.jar file can be found in the ***target*** directory afterwards.

How to use

To explain how JMLS work we will take a look on a small example. The example with sources can be found in this project too (***JMLSExample.jar***). The example demonstrates how to use a few GUI components, how to add a JFrame and a JTabbedPane and how to automatically generate a language menu. To demonstrate how to add icons some components got an icon too. The icons are taken from famfamfam Silk Icons.



instantiate

To instantiate JMLS just provide the path inside the jar file to the language file without extension. The extension will be set automatically depending on the used locale. As 2nd argument provide the default locale which should be used. In this case it is english. JMLS will search for the file Example.en_US in the view folder inside the jar file. If you want to use another language as default you could for example use Locale.GERMANY. In this case JMLS would serach for the file Example.de_DE inside the jar file. More about language files later.
It is recommended to declare MLS as an attribute.

```
MLS mls;
mls = new MLS("view/Example", Locale.US);
mls.setToolTipDuration(-1);
```

Additionally you can decide how long a tooltip should be shown. Per default the tooltip will be fade away after a few seconds even if the mouse isn't moved. Using the `setToolTipDuration(...)` method you can decide how long a tooltip should be there until it fades away.

generate components

To register a component to JMLS it will be generated by JMLS by providing an identifier which will be used in the language file and if necessary some other information. Here are some examples how to generate some components:

```
JButton aButton = mls.generateJButton("aButton");
```

```
JPanel aTitledPanel = mls.generateTitledPanel("aTitledPanel");
JTextField aTextField = mls.generateJTextField("aTextField", true, false, 10
, "some text");
JEditorPane anEditorPane = mls.generateJEditorPane("anEditorPane", true, tru
e, "some text too", "html/plain");
JCheckBox aCheckBox = mls.generateJCheckBox("aCheckBox", true);
```

The first parameter is always the *identifier*. JMLS uses the *identifier* to filter all information for this component out of the language file. As you can see in the example above some components also allow to provide some more information. For example a `JCheckBox` can be set checked. A `JEditorPane` can set editiable and enabled, can get a default text and content type.

language file

As mentioned before JMLS gets all information about a component using its *identifier* from the specific language file. An *identifier* should only contain letters and numbers. Each component can have up to 9 components:

```
[identifier].label=
[identifier].labelJar=
[identifier].labelHDD=
[identifier].icon=
[identifier].shortcut=
[identifier].actionCommand=
[identifier].toolTip=
[identifier].toolTipJar=
[identifier].toolTipHDD=
```

the *label* is the label of a component. For example the shown text on a JButton or a JLabel. Only one label parameter (*label*, *labelJar* or *labelHDD*) is allowed to use. If multiple label parameters are used, the last one will be used by JMLS. If *label* is used the given text will be taken as a label. If *labelJar* is used, a path to the file inside the jar file should be given. The shown text is the text given in the provided file. This is useful for longer texts like instructions. The path has to be given in the form path/to/the/file (do **not** start the path with a '/'). The *labelHDD* parameter is similiar to *labelJar*. But instead of taking the file out of the jar file it uses the file from the harddisk. The path can be given in absolute or relative form.

The *icon* parameter takes a path to an image file inside the jar file. This image will be set as the icon to the component. The path has the form path/to/the/file (do **not** start the path with a '/').

The *shortcut* parameter is the used mnemonic for the component. The given mnemonic should be just one character. If more than one character is given, the first character will be used.

The *actionCommand* is the given actionCommand for this component. The actionCommand can also be given in the constructor of the components. If the actionCommand is given in the constructor, the actionCommand from the language file will be ignored.

The *toolTip* is the shown tooltip for the given component. There are 3 ways to set a tooltip: *toolTip*, *toolTipJar*, *toolTipHDD*. The meanings are the same as for *label*.

Some parameters like *toolTip* or *shortcut* are optional. If no text or path is given (or in other words the value is empty) it will be treated as it wouldn't be there. Therefore it is the same if a parameter is given but empty and if the parameter

isn't given in the language file at all.
JMLS warns you if not even one parameter for an *identifier* is given. In this case just provide an identifier but leave the value empty. For example this is useful for `JList` or `JTextField`. Just provide

```
[identifier].toolTip=
```

and leave it empty and you are good.

JMLS also offers the possibility to leave a message which doesn't belong to a component. For example if you want to give a message to the user that a file couldn't be found, you could write the following line in the language file:

```
fileNotFound=File couldn't be found.
```

Now you can get the text for the identifier by using the following method:

```
mls.getMessage("fileNotFound")
```

JMLS decides depending on the current locale from which language file the message should be taken.
You could also use `fileNotFound.label=File couldn't be found.`. The *label* parameter for simple messages is optional.

The following table shows which component supports which parameter:

Component	**[id].label**	**[id].icon**	**[id].shortcut**	**[id].toolTip**	**[id].actionCommand**
addCustomPanel				x	
addJFrame	x				
addTab	x	x	x	x	
JButton	x	x	x	x	x
JCheckBox	x	x	x	x	x
JCheckBoxMenuItem	x	x	x	x	x
JComboBox	x	x	x	x	x
JEditorPane				x	
JFormattedTextField				x	
JLabel	x	x		x	
JList				x	
JMenu	x	x	x	x	x
JMenuBar				x	
JMenuItem	x	x	x	x	x

JPanel				x	
JPasswordField				x	
JPopupMenu	x			x	
JRadioButton	x	x	x	x	x
JSlider				x	
JTabbedPane				x	
JTable				x	
JTextField				x	
JTextArea				x	
JTextPane				x	
JToggleButton	x	x	x	x	x
TitledPanel	x			x	
JTree				x	

generate language menu

To generate a language menu you just have to write a config file which has to be located inside the jar file and set this path to JMLS. The config path is similiar to the language file and contains up to 2 components per language:

```
[locale]=[language name]
[locale].icon=
```

The *icon* parameter is optional.
For example if you want to support 2 languages and both languages should have an icon flag next to its name, this file should look like this:

```
en_US=english
en_US.icon=path/to/english.png
de_DE=german
de_DE.icon=path/to/german.png
```

Now you just have to set the path to the config file to JMLS:

```
mls.setConfigPath("path/to/configFile");
```

Now you can use this to generate the language menu. The language menu can be generated with `JCheckBoxMenuItems` or `JRadioButtonMenuItems`. In this case we generate a language menu using JCheckBoxMenuItems:

```
mls.addCheckBoxLanguageMenuItem(languageMenu, Locale.US);
```

The first parameter is the MenuItem to which the language menu should be added, and the 2nd parameter is the current locale which should be chosen as the default language. The menu item will switch the language for every registered component for this `MLS` object.
Optionally a `LanguageMenuExtension` can be given as 3rd parameter. This is necessary if for example sub panels use its own `MLS` object. In this case you can call the `translate()` method manually for each sub panel. This could look like this:

```
mls.addCheckBoxLanguageMenuItem(languageMenu, Locale.US, new LanguageMenuExtension() {
    @Override
    public void changeLanguage(Locale lang) {
        aSubPanel.setLanguage(lang);
    }
});
```

Copyright