Internationalization

Translations to Specific languages

- Chinese (Traditional) Translation
- Chinese (Simplified) Translation
- French Localization
- Japanese Localization
- Polish Localization
- Spanish Translation
- Brazilian Portuguese Translation

What developers need to know/do

Jenkins' i18n support has some aspects that developers need to be aware of.

1. Generation of type-safe `Messages` classes from `Messages.properties`.
2. Marking messages in jelly files.
3. Creation of html files for help tips.

Generation of `Messages` classes

Jenkins uses localizer to generate classes for accessing message resources in a type-safe way. For each `src/main/resources/**/Messages.properties`, a Messages class is generated. See the referenced page for how this class looks like. If your IDE fails to find these classes, manually add "target/generated-sources/localizer" directory to your source root.

Wherever the code returns a String for display purpose (such as `Descriptor.getDisplayName()`), use the Messages class to obtain a localized message. At the runtime, a proper locale is selected automatically. A typical workflow for this is as follows:

1. You identify messages that need be localized
2. You place that in Messages.properties. One can choose to have this file for each package, or you could just have one such file for the whole module/plugin.
3. You run `mvn compile` once to re-generate `Messages.java`
4. Update your code to use the newly generated message formatting method

> ✅ As usual, looking at how the core code does this might help you get the idea of how to do this.

> ✅ **Note:** If your message contains single quote character ('), then you need to escape it with another single quote character (i.e. ''). So "that's" becomes "that''s". If you're in need of an apostrophe character you might consider using ' - the unicode character U+2019, written in properties file as \u2019.

Marking messages in jelly files

Your jelly files in `src/main/resources` often contain messages that need to be localized, and those need to be marked as well.

In the simplest case, suppose you have a part of a Jelly file that looks like the following:

```
<h1>Output</h1>
<p>...</p>
```

Then all you need to do is to change this to the following:

```
<h1>${%Output}</h1>
<p>...</p>
```

The ${%...} marker indicates stapler to look for localized resources, and when none is found it'll just print "Output" for this, which is what you want.

Let's consider the case where the localization requires parameters. Suppose you have a Jelly file `foo.jelly` like this:

```
<p>Click <a href="${obj.someMethod(a,b,c)}" >here</a></p>
```

In this case, first you have to write `foo.properties` for the default locale:

```
click.here.link=Click <a href="{0}" >here</a>
```

Then update `foo.jelly` to refer to this like this:

```
<p>${%click.here.link(obj.someMethod(a,b,c))}</p>
```

If you have multiple parameters you can pass them by separating ',' (just like a function call), and from property files you can reference them as {0}, {1}, etc., by following the MessageFormat class convention.

Now consider the case where you put a message in an expression like this. Suppose you have a Jelly file like this:

```
<p>${h.ifThenElse(x,"no error","error")}</p>
```

You can mark those two strings for localization like this:

```
<p>${h.ifThenElse(x,"%no error","%error")}</p>
```

Finally, because you can use expressions in Jelly, you need to ensure your regular strings don't contain brackets.
For example, this will **not** parse correctly, as it will be parsed as an expression:

```
<h1>${%Path to file (.ipa or .apk)}</h1>
```

You need to create two separate strings in this case:

```
<h1>${%Path to file} (${%.ipa or .apk})</h1>
```

Creation of html files for help tips.

It is possible in jelly files to configure entries which can show a tip message when the user clicks on a help icon.

```
<f:entry title="${%title}" help="/plugin/myPlugin/help.html">
```

Then, just create the file help.html for English or help_xx.html for the 'xx' language.

What translators need to know/do

The Jenkins project always welcomes contributions to translations. If you are interested in helping us, just file a pull request on GitHub. In the remainder of this section, we'll discuss what needs to be translated and how.

⚠ Remember that properties files must be encoded in ISO 8859-1. This should be the default anyway.

Translation tools

- Translation Tool : Command line tool to help translators to generate files, add missing keys, and more.
- Translation Assistance Plugin : Adds a dialog box in Jenkins to translate and send missing keys.

Translating `Messages.properties`

Developers place messages that require localizations in `Messages.properties`. Those need to be translated in the usual manner. See `Messages_ja.properties` in the core as an example if you are new to this process.

Sometimes looking at `Messages.properties` alone doesn't give you enough contextual information as to where the messages are used. For this, developers are encouraged to access messages by using the type-safe `Messages` class generated by localizer. To find out where messages are actually used, use your IDE to find all the usages of the message format method.

Translating message references in Jelly

The other messages that need to be translated are in Jelly view files, which are in `src/main/resources/**.jelly`. To localize them, first you run Maven to generate skeleton property file for your locale:

```
$ cd jenkins/core   (or a plugin dir)
$ mvn stapler:i18n -Dlocale=fr
```

This will generate a bunch of `*_fr.properties` all over `src/main/resources` with an empty value. If the file already exists, it will append missing entries to existing files.

You then need to work on each such property file and translate messages. You don't have to translate the entire file — if you leave some entries empty, they'll fall back to the default locale.

✓ Quick locale switcher firefox extension is useful to toggle between various locales.

Translating static HTML resources

Stand-alone HTML files are often used in Jenkins for things like inline help messages. These resources need to be translated by adding the locale code between the file name and the extension. For example, the Japanese version of `abc.html` would be `abc_ja.html`, and British version of it could be `abc_en_GB.html`. These files need to be encoded in UTF-8.

Pushing changes

Once you made some changes, you can commit them. Translators should consider themselves as owning property files for their locale, so feel free to go ahead and just commit. If you are new to this, doing a small commit first is a good idea. You can also always send in a patch if you prefer to be safe.

When starting a translation, try to check if there's anyone else working on the same locale. You can find out who they are by finding existing localization and looking at its history. Try to get in touch with them to avoid a surprise.

Stapler plugin for IntelliJ IDEA

Refactoring the existing code to handle i18n correctly is tedious. So IntelliJ IDEA plugin for Stapler is developed to simplify this (note that JetBrains kindly offered the open-source license for the Jenkins project, so contributers can get the license for free — contact Kohsuke if you need one.)

Once installed, this adds a menu item in the main menu, under "Refactor." It is highly recommended to give some keyboard short cut to this. I use "Ctrl+Alt+I":

```
Refactor  Build  Run  Tools  Version Control  Wind
    Rename...                        Shift+F6
    Change Signature...              Ctrl+F6
    Make Static...                                    odeMo
    Convert To Instance Method...                     paceC
                                                      nitor.j
    Move...                              F6
    Copy...                              F5
    Safe Delete...                  Alt+Delete
    Extract Method...               Ctrl+Alt+M
    Replace Method Code Duplicates...
    Invert Boolean...
    Introduce Variable...           Ctrl+Alt+V
    Introduce Field...              Ctrl+Alt+F    ser
    Introduce Constant...           Ctrl+Alt+C    NDS
    Introduce Parameter...          Ctrl+Alt+P    NDS
    Extract Interface...
    Extract Superclass...
    Use Interface Where Possible...
    Pull Members Up...
    Push Members Down...                          utSt
    Replace Inheritance with Delegation...        ;
    Inline...                       Ctrl+Alt+N
    Convert Anonymous to Inner...
    Encapsulate Fields...                         enBu
    Replace Temp with Query...
    Replace Constructor with Factory Method...
    Generify...                                   enSt
    Migrate...
    Extract Include File...
    Internationalize...
    i18n for stapler                Ctrl+Alt+I    t en
        rangeCheck(start, field);
        rangeCheck(end, field);
```
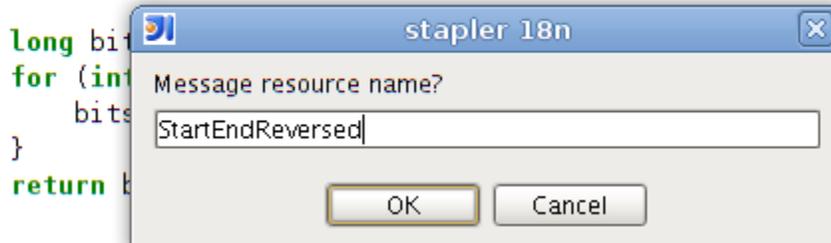
Now, to use this feature, select the message you'd like to internationalize, then trigger this refactoring command. It'll ask you the message property name for this, so pick a name:

```
if (step <= 0)
    error("step must be positive, but found " + step);
if (start>end)
    error("You mean "+end+"-"+start+"?");

long bit
for (int
    bits
}
return b
```

stapler 18n

Message resource name?

StartEndReversed

OK   Cancel

IDEA will place the message into the resource file, and adjust the code accordingly. Note that an error is highlighted until you run `mvn compile` again to generate new methods on the `Messages` class.

```
if (step <= 0)
    error("step must be positive, but found " + step);
if (start>end)
    error(Messages.BaseParser_StartEndReversed(end,start));
```

Stapler plugin for NetBeans

See NetBeans plugin for Stapler for details.

Translation Completeness Reports

Visit http://www.simonwiest.de/glottr/report/ for Glottr Report by Unknown User (swiest)