Development

JT Harness Developer's Guide

This document helps a new developer understand the JT harness source code.

Table of Contents

Basic Conventions

Source files are text files in ISO8859-1 format, always in English. Tab characters are defined as eight spaces. The standard indentation unit is four spaces. It is common to configure your editor to insert four spaces for each press of the Tab key and substitute a tab character for every two of those key presses. *Only use space characters for indentation.*

Eighty character lines are not required. It is best to keep lines under 80 columns if possible, although longer lines are acceptable if it makes the code more readable. It is common to keep a long literal string on a single line and go to a new line at the next opportunity, usually for the next parameter. *Do not allow your editor to reformat all the code in a file to eighty columns because of word wrapping.* Make your own decision as to whether lines should be wrapped for display purposes.

Method definitions and calls do not have a space between the method name and the parentheses that surround the argument list. For conditional statements there is a space after the keyword and before the condition list (`while`, `for`, `if`, ...) Curley brackets begin at the end of the line that defines the context and end on their own line, aligned to the left justification of the beginning context line. Comment the ending brackets when there is a possibility for confusion. This comment goes at the next four character indent for that line (see the example below).

```
Example:

public void exampleMethod(boolean foo) {
    if (foo) {
                while (x) {
                    x = method1(foo, bar);
                }        // while
    }
}   // exampleMethod()
```

Internationalization

All code in the core of the JT harness and utilities that may be distributed externally must be internationalized. For the majority of development work, this just means that literal strings must be retrieved from a resource bundle. This applies to strings being written for end user debugging, file output (not data files), warning and error messages, and all strings in the GUI.

It is important to follow the established coding patterns maintain consistency in the code and to facilitate automated string checking in the build (discussed below).

To retrieve a literal string from a resource bundle (usually in non-GUI code) use this coding pattern:

```
import com.sun.javatest.util.I18NResourceBundle;

class .... {
    public String exMethod() {
        return i18n.getString("allTestsFilter.name");
    }

    private static I18NResourceBundle i18n = I18NResourceBundle.getBundleForClass(Harness.class);
}
```

The `I18NResourceBundle` class knows how to locate the correct resource bundle at runtime. It is static because the bundle is static once the correct bundle has been located (determined at runtime). Strings are always located in the default locale bundle `i18n.properties`. There is one bundle for each package. The `I18NResourceBundle`class finds the right bundle for you based on the class you specify.

The name space inside the bundle is up to you. For example, the prefix for the `BranchPanel` class is `bp.*`. In the code above, `allTestsFilter` is the prefix for strings used by the `AllTestsFilter` class. See the existing source code in the workspace for more examples. You generally only need to decide this when you create a new class.

Note that the build has an I18N test that verifies that all keys referenced by calls to `i18n.getI18NString()` exist in the `i18n.properties` bundle. It also does the reverse check to see if there are orphan entries in the bundle. These are both fatal build errors, which is why running the `jt-qc` build target is important.

Writing GUI code takes this procedure a bit further, especially when combined with Section 508 requirements. This is described in GUI Coding for I18N and 508 Compliance.

Section 508

There is very little for you to do to be Section 508 compliant when you are not writing GUI code. Remember that when you emit HTML code, it must be 508 compliant. Most of the 508 complexity in the JT harness is for GUI code as described in GUI Coding for I18N and 508 Compliance.

GUI Coding for I18N and 508 Compliance

To comply with i18n and Section 508 requirements, leverage the `UIFactory`. This class has methods to retrieve all sorts of properly constructed GUI components. For example, the following example shows how to create a button:

```
button = uif.createButton("exec.wd.open", optionListener),
```

This call returns a button with internationalized text, an accessible description and an accessible name. It also attaches the given action listener for you. There are often multiple versions of each `create*` method to accomodate common needs - for example, attaching listeners, and defining icons. Often you will need to look at the code in the `UIFactory`class to actually understand what things to. This is not documented extensively in the comments generated by the Javadoc™ tool because it is not a public API.

In addition to methods that create all the usual objects, you will also find methdods to create the following objects:

- Titled borders
- Glue and strut components
- Basic dialog boxes
- JPanel
- Read-only text fields
- Scrolled panes

Note: Make maximum effort to use the `UIFactory` whenever possible, it contains many workarounds and good coding practices. It also makes it possible to make mass changes in a central location when needed. For example, working around a bug in a `JTabbedPane`.

The `UIFactory` (`uif` is the example above) uses the same resource bundle as discussed above for internationalization, but the you interact with `UIFactory` instead of `I18NResourceBundle` when writing GUI code. Usually, you pass around an existing instance of the factory, rather than create a new one. Notice that the constructor for most the JT harness GUI classes require a factory as a parameter.

When using the `createButton` method, each `create*` method may require you to put slightly different resource entries in the resource bundle. In the example above, `exec.wd.open` is only a prefix, the resource bundle contains many more entries with that prefix. For a button, you supply the following in the bundle:

```
exec.wd.open.btn
    The string that appears on the button itself. The .btn extension is exclusive to buttons.
exec.wd.open.tip
    The tooltip for this button. The .tip extension is always used for tooltips. Most components require this.
exec.wd.open.mne
    The mnemonic (keystroke) to trigger the button. Defined as 'O' in this case. This is generally required for
buttons, labels, and menu components and is a 508 requirement that improves usability.
```

Be sure to look in `UIFactory` or its JavaDoc tool comments to find out what is required for each type of component you create. The other two common suffixes are:

```
*.desc
    The acccessible (508) description for the component. Most components require this.
*.name
    The acccessible (508) name for the component. Most components require this.
```

The build automatically checks for missing attributes. If you forget to put something in the resource bundle it notifies you and stops the build.

There quite a few special cases that are not described here. If you have a question about your situation it is best to check with other developers. Also, the `UIFactory` is not necessarily complete, if you think a new method or variation would be useful, mail the other developers with your idea.

Common Errors

The most common error is forgetting to add an entry to the resource file for GUI components. The build will catch this problem and notify you.