jInfer: Dealing with jInfer and NetBeans Platform

Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, Matej Vitásek
Advisors: RNDr. Irena Mlýnková, Ph.D., Martin Nečaský, Ph.D.

Praha, 2011

Dealing with jInfer and NetBeans Platform

Target audience: jInfer developers. Anyone who needs to interact with jInfer or NetBeans while hacking our framework.

This tutorial deals with a few important but pretty specific parts of interface between your module (or any logic you implement) and either jInfer or NetBeans Platform. Not nearly everything regarding interaction with NBP is mentioned here, please refer to the relevant FAQ.

This tutorial assumes that you are a seasoned Java developer. Having experience with programming in some kind of framework (NetBeans Platform above all) will help you a lot. Make sure you have read the article on architecture, data structures and inference process to understand what you will be implementing. Having read the documentation for remaining modules will help you too. Also, before starting this tutorial, make sure you can build jInfer from sources.

Overview

1. Module visibility
2. Error handling
3. Interruptions
4. Dialogs
5. Configuration - options, preferences
6. Rule display
7. Console output, logging
8. RunningProject class
9. Module selection

Module visibility

Each NBP module allows to set some spackages as public, which means their content will be available to other modules. That means, if module *A* declares a dependency on module *B*, it will be able to use only classes from those packages that were set as public in module *B*.

Important notice: *Public packages do not affect Lookup mechanism, i.e. class annotated with* `@ServiceProvider` *does not need to be in a public package in order to be looked up.*

There are two ways to set packages visible, first is to set it manually in module's `project.xml` file located in *ModulePath/nbproject/* folder.

Example:

```
// This is a portion of example project.xml file.
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="\url{http://www.netbeans.org/ns/project/1">}
    <type>org.netbeans.modules.apisupport.project</type>
    <configuration>
        <data xmlns="\url{http://www.netbeans.org/ns/nb-module-project/3">}
            ....
            <public-packages>
                <package>some.example.package.name</package>
            </public-packages>
        </data>
    </configuration>
</project>
```

Another way to set public packages is through GUI:

1. Right-click the module, select *Properties*.
2. In *API Versioning*, there is a *Public Packages* section.
3. Check all packages that should be public.

Error handling

Each run of inference is encapsulated in a `try-catch` block, so it is safe to throw any exception in the inference process. Each thrown exception will be caught, get logged, presented to the user and inference will stop. However if module uses threads which throw exceptions, it is caught by NBP, instead of our code. Because of this its reasonable to catch this exception in your module and re-throw it in the right thread.

Interruptions

In jInfer user can stop the inference at any moment of its run. For this reason, modules have to check in every time-consuming place (such as long loops) whether this is the case. Example below shows how to check the interruption from user and what to do to correctly stop the inference.

Example:

```
for (forever) {
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
    doStuff();
}
```

Dialogs

Creation of open/save dialogs or message windows is done using standard NBP API. For dialogs you can use the standard Java `JFileChooser`, but NBP `FileChooserBuilder` is more convenient. This class creates dialog which remembers last-used directory according to a key passed into its constructor.

Example:

```
// The default dir to use if no value is stored
File home = new File(System.getProperty("user.home"));
// Now build a file chooser and invoke the dialog in one line of code
File toAdd = new FileChooserBuilder(CallingClass.class)
                    .setTitle("Some dialog title")
                    .setDefaultWorkingDirectory(home).showOpenDialog();
// Result will be null if the user clicked cancel or closed the dialog w/o OK
if (toAdd != null) {
    //do something
}
```

For message windows the `DialogDisplayer` NBP API class is used. To determine which type of message will be shown (Confirmation dialog, message dialog...), `NotifyDescriptor` is used. Following is a small example, which creates standard message notification. For more information please look at NBP Dialogs API FAQ.

Example:

```
// Creates standard information message with text "Hello world"
NotifyDescriptor nd = new NotifyDescriptor
                          .Message("Hello world",
                             NotifyDescriptor.INFORMATION_MESSAGE);
DialogDisplayer.getDefault().notify(nd);
```

Configuration - options, preferences

Option panels located in *Tools* > *Options* menu are a standard part of NBP API. We created a *jInfer* option sub-category to place panels with configuration valid across all jInfer projects. Description of the few step to create your own option panel follows. For more information, please visit NetBeans Options tutorial.

1. Create options window:
   - Right click *Source Packages* in your module, select *New* > *Other*.
   - Under Categories, select Module Development. Under File Types, select Options Panel. Click *Next*.
   - Keep checked *Create Secondary Panel* and choose jInfer as the *Primary Panel*. Fill in remaining fields. Click *Next*.
   - Fill in *Class Name Prefix* and *Package* and click *Finish*.
2. Design newly created *ClassNamePrefix*Panel.java.
3. Implement `store()` and `load()` methods according to comments inside.

Both `store()` and `load()` methods use `NbPreferences` class. Example below shows how to store and load Preferences.

Example:

```
public void store() {
  // Saves the state of checkBox into Preferences for ExampleClass
  // class in property with name "PropertyName"
  NbPreferences.forModule(ExampleClass.class)
      .putBoolean("PropertyName", someCheckBox.isSelected());
}
public void load() {
  // Get from ExampleClass class preferences the property with name
  // "PropertyName". Second parameter of method getBoolean is used
  // as default if no property with defined name is saved in preferences.
  someCheckBox.setSelected(NbPreferences.forModule(ExampleClass.class)
                              .getBoolean("PropertyName", true));
}
```

Second type of preferences used in jInfer is Project properties. Each jInfer project has its properties panel accessible in its content menu and these properties are applied for each project separately. To implement this kind of preferences, it is necessary to implement `PropertiesPanelProvider` interface and extend the `AbstractPropertiesPanel` class. Each Project properties window has category tree, where each category represents a separate panel with properties. This category is declared in the provider interface, properties panel itself is defined in class extending `AbstractPropertiesPanel`.

`PropertiesPanelProvider` As was mentioned above, `PropertiesPanelProvider` defines category in Project properties windows. In the example below, we'll try to explain how to create a simple provider and what each method is responsible for.

Example:

```
public class ExampleProvider implemenets PropertiesPanelProvider {
  // Programmatic name of this provider.
  public String getName() {
    return "ExampleProvider";
  }
  // User-friendly name - this name will be displayed in category tree.
  public String getDisplayName() {
    return "Example Category";
  }
  // Priority of category. Higher the number, higher
  // will the category be in the tree. If two categories have
  // same priorities, they are sorted according
  // to their names.
  public int getPriority() {
    return 0;
  }
  // Returns properties panel defined for this category.
  public AbstractPropertiesPanel getPanel(final Properties properties) {
    return new ExamplePropertiesPanel(properties);
  }

  // Defines parent category of this category. If null is returned, this is top
  // level category. In other cases id (programmatic name) of parent category
  // must be returned.
  public String getParent() {
    return null;
  }

  // Optional. For each category, list of virtual categories can be defined.
  // Instead of properties panel, VirtualCategoryPanel only informs of type and
  // installed number of modules of a certain type.
  public List getSubCategories() {
    return null;
  }
}
```

`AbstractPropertiesPanel` This class represents the visual component of properties category. You can design it in whatever way you like. For proper functionality, a few steps must be followed.

1. Call `super(Properties)` in constructor: this causes `Properties` instance to be saved into `properties` protected field and you can use it in `load()` and `store()` methods.
2. Implement `load()` method: In this method values previously saved into `Properties` instance provided to this panel are loaded into components in this panel.
3. Implement `store()` method: Here the values gathered from components in this panel are saved into `Properties`.

Example:

```
// This is only a part of the actual class...
public class ExamplePanel extends AbstractPropertiesPanel {

  public ExamplePanel(final Properties properties) {
    super(properties);
  }

  public abstract void store() {
    properties.setProperty("exampleKey", someTextField.getText());
```

```
  }

  public abstract void load() {
    String propValue = properties.getProperty("exampleKey", "default value");
    someTextField.setText(propValue);
  }
}
```

Rule display

Rule displayer is a component used for visualization of rules in any step of inference process. If rule displayers bundled with jInfer are insufficient, you can implement a rule displayer on your own. You just need to follow these few easy steps.

1. Implement `cz.cuni.mff.ksi.jinfer.base.interfaces.RuleDisplayer`
   - `RuleDisplayer` interface extends `NamedModule`, so you need to override its methods too.
   - Implement `createDisplayer(name, rules)` method which creates displayer window responsible for displaying rules.
2. Annotate this class with `@ServiceProvider(service = RuleDisplayer.class)`.

Important notice: never forget to clone the rules you want to display. Rule displayer usually runs in its own thread, and accessing the rules that are being simplified at the same time might lead to weird results. You can use `CloneHelper` to clone the rules.

Example:

```
@ServiceProvider(service = RuleDisplayer.class)
public final class ExampleRuleDisplayer implements RuleDisplayer {

  @Override
  public void createDisplayer(String panelName, List rules) {
    // Creates standard NBP TopComponent in which are rules displayed.
    ExampleRDTopComp topComponent = ExampleRDTopComp.findInstance();
    if (!topComponent.isOpened()) {
      topComponent.open();
    }
    topComponent.createRuleDisplayer(panelName, rules);
  }

  @Override
  public String getName() {
    return "ExampleDisplayer";
  }

  @Override
  public String getDisplayName() {
    return "Example rule displayer";
  }

  @Override
  public String getModuleDescription() {
    return getDisplayName();
  }
}
```

Example of invocation with cloning:

```
RuleDisplayerHelper.showRulesAsync("Panel name",
    new CloneHelper().cloneGrammar(grammar), true);
```

Console output, logging

To print to console output, NBP provides `IOProvider` and `InputOutput` classes. `IOProvider`'s method `getIO(String, boolean)` returns `InputOutput` instance, which can be used to obtain `Reader`/`OutputWriter` to read/write into output window.

Example:

```
// Creates new output window, where first parameter is a name and second
// is flag determining whether a new window should be created even if there is
//  already a window with the same name.
InputOutput ioResult = IOProvider.getDefault().getIO("Example", true);
ioResult.getOut().println("Hello world");

// After writing everything into output window, close the output.
ioResult.getOut().close();
```

Logging is done in jInfer using Log4j tool. If you are not familiar with Log4j, example below shows simple usage. All logged messages are saved in a standard log file placed in *UserHome/.jinfer/* folder and printed into "jInfer" output window in the application. For each of these two places a log level can be set in jInfer options.

Example:

```
// Creates Logger instance for this particular class
private static final Logger LOG = Logger.getLogger(Example.class);

    ...

// Log a message with info level
LOG.info("Some message with info log level");
// Log an error message
LOG.error("OMG, error occurred!");
```

`RunningProject` class

This class provides access to information regarding currently running inference. It is a singleton, corresponding with the fact that at a given time, at most one inference (project) can be running in the whole NB. Consequently, all relevant methods are synchronized. NB project corresponding to the running inference can be retrieved by calling `getActiveProject()`. Its properties can be retrieved by calling `getActiveProjectProps()`. This method will work also if there is no running project - it just retrieves an empty instance of `Preferences`, which is useful for example in JUnit tests.

Example code retrieving properties for DTD export:

```
Properties properties = RunningProject.getActiveProjectProps(
                            DTDExportPropertiesPanel.NAME);
```

Aside from keeping information about the currently running project, this class keeps information about the capabilities of the following module in the inference chain. This can be retrieved by calling the `getNextModuleCaps()` method. Again, if this information is not available, empty `Capabilities` instance will be returned.

Example code retrieving *Simplifier* capabilities in *BasicIGG*:

```
if (!RunningProject.getNextModuleCaps().getCapabilities().contains(
            Capabilities.CAN_HANDLE_COMPLEX_REGEXPS)) {
  ... expand rules ...
}
```

Module selection

This part of the tutorial will show the full process of implementing a choice between two submodules in an inference module, for example simplifier. Similar principles apply one level higher, when dealing with inference modules themselves.

Interface

First of all, we need an interface encapsulating the new submodule. It has to extend `NamedModule` interface.

```
public interface Foo extends NamedModule {

  String bar();
}
```

Implementations

Second, we need some implementations of this interface. At least two, otherwise it's boring. They will have to implement all required methods and register themselves as service providers for `Foo`.

Example of the first implementation. The second will be the same (just change "ones" to "twos").

```
@ServiceProvider(service = Foo.class)
public class Impl1 implements Foo {

  // NamedModule stuff

  @Override
  public String getName() {
    return "Impl1"; // unix-style name
  }

  @Override
  public String getDisplayName() {
    return "Foo implementation number 1"; // user-friendly name
  }

  @Override
  public String getModuleDescription() {
    return getDisplayName(); // irrelevant now
  }

  // logic from Foo

  @Override
  public String bar() {
    return "Hello world from Impl1!";
  }
}
```

Properties panel

Now we have to create a properties panel. Refer to an earlier part of this document for general reference. Here we assume a combo box already present on the properties panel, with the correct renderer set (`ProjectPropsComboRenderer`). We need to do 2 things: while loading the properties panel, fill it with names of all implementations and select the currently chosen one (or default). While saving, we have to get the name of the selected implementation and save it.

Example of panel load code:

```
@Override
public final void load() {
  combo.setModel(new DefaultComboBoxModel(
          ModuleSelectionHelper.lookupImpls(Foo.class).toArray()));

  combo.setSelectedItem(ModuleSelectionHelper.lookupImpl(Foo.class,
                  properties.getProperty(
                     "selected.foo.implementation", // property name
```

```
                    "Impl1"))); // default property value
                                        // - same as Impl1.getName()!
}
```

Example of panel save code:

```
@Override
public void store() {
  properties.setProperty("selected.foo.implementation",
          ((NamedModule) combo.getSelectedItem()).getName());
}
```

Example of combo box renderer setting:

```
combo.setRenderer(new ProjectPropsComboRenderer(combo.getRenderer()));
```

Invocation

Finally, there will be a place (in our example simplifier) where we actually want to load the properties of the running project and lookup the selected implementation of `Foo`.

Example lookup and usage:

```
Properties p = RunningProject.
    getActiveProjectProps("ExampleProvider"); // the name provided in
                                              // properties panel provider

Foo foo = ModuleSelectionHelper.lookupImpl(Foo.class,
    p.getProperty("selected.foo.implementation", "Impl1"));

println(foo.bar());
```

A few things should be noted:

- Module names (`getName()`) have to be unique.
- It is a good practice to store property names and default values as constants, for example in super module - do a reference search in our code to find out how we do it.
- `getUserDescription()` should work in modules that have submodules as follows: it should somehow return the name of the module and names of all selected submodules to indicate inner workings of this module. For example *My Simplifier (Clustering Logic A, Regexper #3)*.