Developer documentation

Wollenburger Antoine

April 7, 2013

Introduction

Here is the brief tutorial about how to make your own plugin for our plateform. //TODO make a real introduction.

Note : All codes listings are given in theirs files forms.

Chapter 1

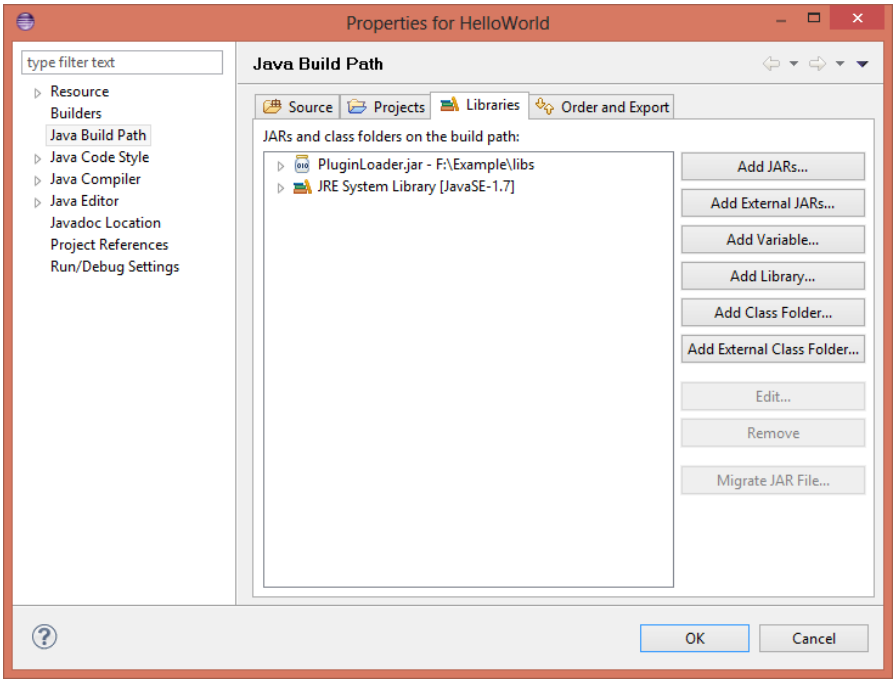
Create your first plugin

The first thing you need is a new project. Name it at your conveniance. For the example, we name it "HelloWorld".

1.1 Fix the Java Build Path

Secondly add "PluginLoader.jar" to the Java Build Path (right clic on th project, Properties, Java Build Path).

Figure 1.1: Fixing Java Build Path



1.2 Saying "Hello World"

Before creating a new class, you must create a new package. In this example we take "com.ornicare.helloworld". Then create a new class and make it an extend of PluginRunnable and create a run method : it's the hook for the platform. It must look like this :

Listing 1.1: MainClass.java

```
package com.ornicare.helloworld;

import com.space.plugin.PluginRunnable;


public class MainClass extends PluginRunnable {

    @Override
    public void run() {
        System.out.println("Hello world !");
    }
```

1.3 Interface your classe

In order to use your plugin in another plugin, you must interface it. In this example, it's very simple, create a new interface "IMainClass" like this :

Listing 1.2: IMainClass.java

```
public interface IMainClass {
}
```

Note : it's necessary when you need to cast another plugin object (beacause it is proxies, you need to use interfaces).

1.4 Generate the propertie file

Create a new file named "plugin.properties" into the project. In this file add :

- The plugin name : "name = HelloWorld"
- The hook's class path : "main = com.ornicare.helloworld.MainClass"
- The runnable attribut : "runnable = true"
- The launchable attribut : "launch = true"

It gave something like this :

Listing 1.3: plugin.properties

```
name = HelloWorld
main = com.ornicare.helloworld.MainClass
runnable = true
launch = true
```

1.5 Exporting your first plugin

To run this, you need to create a jar of your plugin : "File > Export > Java/JAR File". Put the created jar into the plugin directory and run the plugin loader with "java -jar PluginLoader.jar".

See the how to launch the plugin loader in another document.

Chapter 2

Intricated plugins

2.1 Using another plugin classes

For this example, we are going to use the "CConsole" plugin : it's just a plugin to display the java console (and few other things). So put "CConsole.jar" into the plugin directory. Add it to your project Java Build Path. And use it. For example :

Listing 2.1: HelloWorld.java

```
import cconsole.CConsole;

import com.space.plugin.PluginRunnable;

public class MainClass extends PluginRunnable {

    @Override
    public void run() {
        CConsole.load();
        CConsole.println("Hello world !");
    }
}
```

Then you need to add the following line to your properties file : "depend = CConsole".

2.2 Using another plugin objects

2.2.1 Create a content provider

Basically, it's just a plugin not conceive to be runnable. In our example, we are going to use a class which provide some functions on an int[].

Plugin TableHelper

Listing 2.2: TableHelper.java

```
package com.ornicare.tablehelper;

import com.space.plugin.PluginContentProvider;

public class TableHelper extends PluginContentProvider implements
    ITableHelper{

    private int[] tab;

    public TableHelper() {
        this.tab = new int[0];
    }

    public TableHelper(int[] tab) {
        this.tab = tab;
    }

    @Override
    public int sum() {
        int sum = 0;
        for(int i : tab) sum+=i;
        return  sum;
    }

    @Override
    public int max() {
        int max = 0;
        if(tab.length>0) {
            max = tab[0];
            for(int i : tab) max=max>i?max:i;
        }
        return  max;
    }

    @Override
    public int min() {
        int min = 0;
        if(tab.length>0) {
            min = tab[0];
            for(int i : tab) min=min<i?min:i;
        }
        return  min;
    }

    @Override
    public int average() {
        return  tab.length>0?sum()/tab.length:0;
    }
```

```
    @Override
    public double squareType() {
        if(tab.length<0) return 0;
        int sum = 0;
        int ave = average();
        for(int i : tab) sum+=(i-ave)*(i-ave);
        return Math.sqrt(sum/tab.length);
    }

}
```

See how this class is build : it extends PluginContentProvider and implements its own interface. It's necessary to retrieve an object.

Listing 2.3: ITableHelper.java

```
package com.ornicare.tablehelper;

public interface ITableHelper {

    public abstract int sum();

    public abstract int max();

    public abstract int min();

    public abstract int average();

    public abstract double squareType();

}
```

Listing 2.4: ITableHelper.java

```
name = TableHelper
main = com.ornicare.tablehelper.TableHelper
```

For a content provider, main class mean instanciable class.

2.2.2 Plugin TableUser

For the java build path, make like in the first example.

Listing 2.5: MainClass.java

```
package com.ornicare.tableuser;

import com.ornicare.tablehelper.ITableHelper;
import com.space.plugin.PluginRunnable;
```

```java
public class MainClass extends PluginRunnable implements IMainClass {

    @Override
    public void run() {
        //ITableHelper tableau = (ITableHelper)getPluginUsingConstructor("
            Tableau", new Class<?>[]{int.class,int.class,int.class,int.
            class}, 1,2,3,4);
        ITableHelper tableau = (ITableHelper)getPlugin("TableHelper", new
            int[]{1,2,3,4});

        System.out.println("Sum : "+tableau.sum());
        System.out.println("Min : "+tableau.min());
        System.out.println("Max : "+tableau.max());
        System.out.println("Average : "+tableau.average());
        System.out.println("SquareType : "+tableau.squareType());
    }
}
```

Listing 2.6: IMainClass.java

```java
package com.ornicare.tableuser;

public interface IMainClass {

}
```

Listing 2.7: ITableHelper.java

```
name = TableUser
main = com.ornicare.tableuser.MainClass
launch = true
depend = TableHelper
runnable = true
```

You could note the use of the getPlugin function.

2.2.3 Execution

With only this two plusgins, you may obtain the following result :

Listing 2.8: HelloWorld.java

```
F:\GitHub\Lib>java -jar PluginLoader.jar
Plugins folder in use : F:\GitHub\Lib\plugins

Dependencies groups (linked plugins) :
Group 0 :
    TableHelper [jar_name = TableHelper]
    TableUser [jar_name = TableUser]
```

```
Running : TableUser
Sum : 10
Min : 1
Max : 4
Average : 2
SquareType : 1.0

F:\GitHub\Lib>
```

Chapter 3

Make a regular “plugin.properties”

You are allowed to use the following attributes :

- The plugin name : “name = your_plugin_name”
- The hook’s class path : “main = package.hook_class_name”
- The runnable attribut : “runnable = true\false”
- If you want a single instance : “singleton = true\false”
- Make your plugin lazy : “lazy = true\false”
- Indicate dependencies to others plugins : “depend = plugin1, plugin2, ...”
- Launch automatically : “launch = true\false”. The plugin need to be runnable.