<!--QiQu-->

Extending The QiQu Script Language

aloba
Softwareentwicklung + Consulting

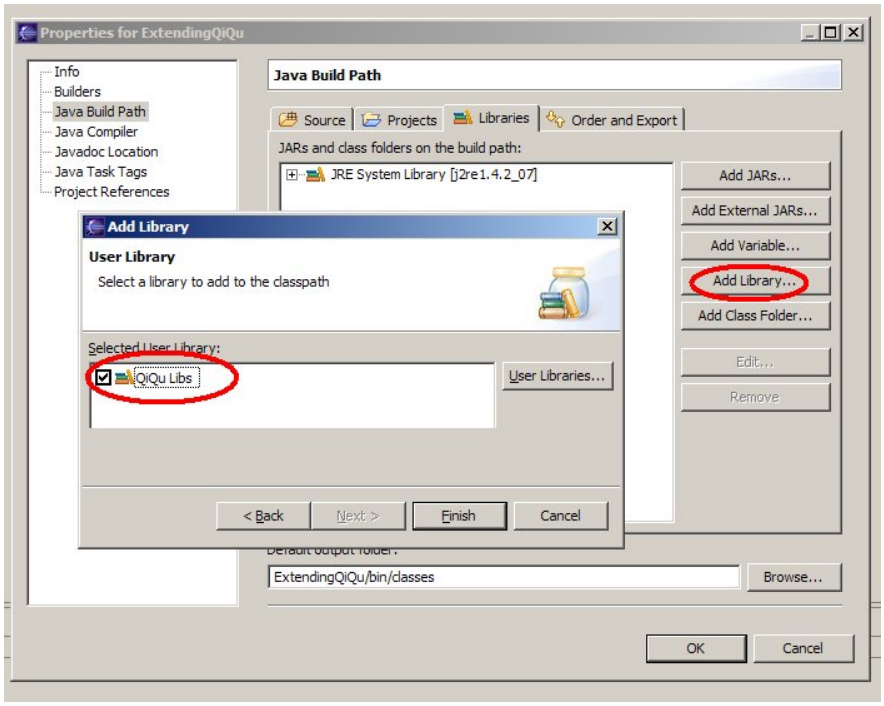
SOURCEFORGE™ .net

Table of Contents

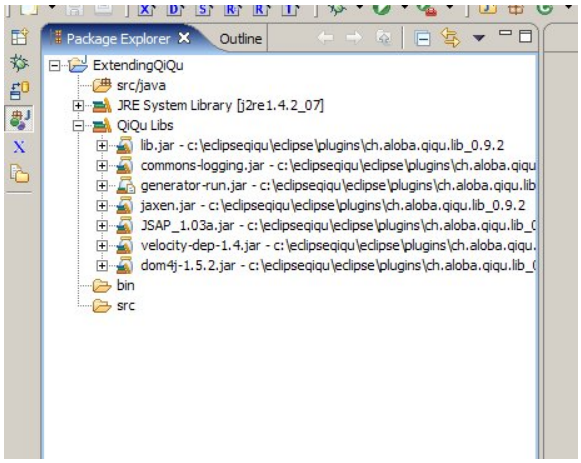
Setting up an Eclipse-Javaproject to extend QiQu

Create a new Java Project in Eclipse. Configure the source and output folders (e.g. source folders "src/java" and "src/javatest"; output folder "bin/classes").

Open the project properties and select "Java Build Path" and choose the register "Libraries". Press the "Add Library..." - button, select "User defined Libraries" and finally check mark "QiQu Libs" entry. This will add all necessary Jar-files to the classpath of your project.



Done that, your project should look somehow like the one in the next picture.



Now, let's move on and ...

Write your first QiQu Command

We will start with a really simple example. Our goal is to write a command with just just one paramter. The content of the paramter has to be displayed in the console output.

```
<Console output="a text"/>
```

So let's create a new java class, call it "Console" and extend it from the class "BaseCommand". Your class should look like the one shown next:

```
/*
 * Created on 07.01.2006
 */
package ch.aloba.qiqu.extend;

import ch.aloba.qiqu.script.base.BaseCommand;

/**
 * A simple command that writes to the output.
 *
 * @author Hansjoerg
 */
public class Console extends BaseCommand
{
}
```

Now we have to implement some required methods. Let's have a look at each one of them.

getCommandName

GetCommandName defines the Name of the command. We would like our command to be known as "Console", so that is the string, this method should return (or alternatively `Console.class.getSimpleName()`).

getDescription

This method returns a short description of the command. It is used to create the documentation and this is as well the text, which is displayed as tooltip in the editor.

getCommandGroup

When creating the documentation, commands with the same groupname are - how else should it be - grouped together. This information is used for nothing else.

Therefore let's add the following lines to our class:

```
/**
 * @see ch.aloba.qiqu.script.base.BaseCommand#getCommandName()
 */
public String getCommandName()
{
        return "Console";
}

/**
 * @see ch.aloba.qiqu.script.base.BaseCommand#getDescription()
```

```
 */
public String getDescription()
{
        return "Writes the content of the attribute 'output' to the console.";
}

/**
 * @see ch.aloba.qiqu.script.base.BaseCommand#getCommandGroup()
 */
public String getCommandGroup()
{
        return "Logging";
}
```

isUsingSubCommand

This method tells the QiQu-System, if this command could have nested commands. Typical examples of such commands are the loop-commands "For" and "While" or the select-command "If". Our simple command "Console" won't have any subcommands, so this method will return false.

setParameters

In the Method setParameters, we must fulfill two chores. The first is to instance all possible parameters of the command and the second is to define all valid combinations of parameters.

In order to instance an parameter, we have to know the type it should have. The "output" parameter has the type `CmdParamText` since it is a plain text parameter. All other possible types will be explained later in this tutorial.

The "BaseCommand" implements factory methods for all possible types. Each of these factory methods takes to input parameters. The first one is the name and the second is a description of the parameter. Again, this description is used to create the documentation and to define the tooltip text in the editor.

In order to have an instance of the parameter, we create an appropriate member variable in the class.

Additionally, we must define the possible combinations of parameters. This is also done by calling the appropriate methods from the superclass: `addParameterCombination()`. Our command has just one valid combination of parameters: "output" is the one and only parameter and it is mandatory. This adds the following lines to our class:

```
private CmdParamText m_outputParam = null;

/**
 * @see ch.aloba.qiqu.script.base.BaseCommand#setParameters()
 */
protected void setParameters()
{
        m_outputParam = addTextParameter("output", "defines the text which will be writ
        addParameterCombination(m_outputParam);
}
```

If there were other possible parameter combinations, we could add them calling `addParameterCombination()` with other parameters.

doIt

Everything is now ready to implement the real logic of our command:

```
/**
 * @see ch.aloba.qiqu.script.base.IBaseCommand#doIt()
 */
public int doIt() throws CommandException
{
        System.out.println(m_outputParam.getValue());
        return IBaseCommand.COMMAND_RETURN_TYPE_NO_LOOP;
}
```

Calling `getValue` on a text returns the text, which is defined in the parameter. Since this text has to be written to the console, we simply make a call to `System.out.printl()`.

Since this command isn't a loop command, the `doIt` method returns the value `IBaseCommand.COMMAND_RETURN_TYPE_NO_LOOP`.

Congratulations, you've just extended QiQu and wrote your first QiQu-Command. Now, let me explain how you can...

Test and Debug an user defined Command

In order to test an debug our command, we need a couple of thing to do:

1. Write a QiQu-library descriptor file
2. Write a QiQu-script that uses the command
3. Write a class, which loads the library into the QiQu-runtime system and launches the QiQu-script

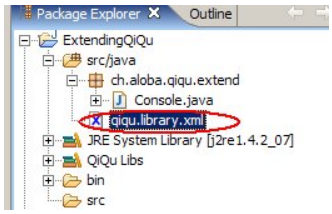
Writing a QiQu-library descriptor file

The QiQu-library descriptor file has to be called "qiqu.library.xml". It must be in the jar containing the commands. It does not have to be in a special directory.

Our QiQu-library descriptor file looks as follows:

```
<QiQuLib>
        <command fqn="ch.aloba.qiqu.extend.Console"/>
</QiQuLib>
```

Let us put the file directly into the root of our source:



Write a QiQu-script that uses the command

Our command does not need anything special, so our script just needs one line:

```
<QiQuScript>
        <Console output="a text"/>
</QiQuScript>
```

Let's call the file "testConsoleCommand.qiq" and add it to the "script" folder of our project. Since the QiQu-script editor doesn't know our command yet, it will display an appropriate marker. I show you in a moment, how you can register the command for the editor as well. For now, ignore this.

Write the testclass

As mentioned, we have to register our library and execute the QiQu-script in the test class. All that is pretty easy and can be done with just three lines. So let us create the following class:

```
package ch.aloba.qiqu.extend;

import ch.aloba.qiqu.librarydata.LibraryManager;
import ch.aloba.qiqu.script.run.QiQuScriptRun;

public class TestConsoleCommand
{
        public static void main(String[] args) throws Exception
```

```
        {
                String[] programargs = new String[] {"-s", "script/testConsoleCommand.q
                QiQuScriptRun runner = new QiQuScriptRun();
                LibraryManager.getInstance().loadLibraryInClasspath("src/java/qiqu.libr
                runner.runScript(programargs);
        }
}
```

I've put the class under a new source-folder called "src/javatest" in order to separate the test classes from the ordinary code.

Let it Run ...

Now go and run the testclass. Somewhere in the output, the defined text in the parameter "output" will appear. Since we are writing directly to the console output, we can't tell exactly where it will appear.

I think I don't have to tell you, that by enabling a breakpoint inside the `doIt`-method of your "Console" command and starting the testclass with a debug launch will initialize a debug session.

Making the new command known to the QiQu-script editor

When we wrote our simple testscript, we noticed the QiQu-script editor is not aware of our new command. This means we don't have any code-assist or tooltips for our command and parameter, and there is also an error marker displayed.

Two steps are necessary to register our command in the QiQu-script editor:

1. Pack command and the QiQu-library descriptor file into a single jar-file
2. Register the jar-file in the preference dialog of the QiQu-script editor

Packing the command and the QiQu-library descriptor into a single jar-file

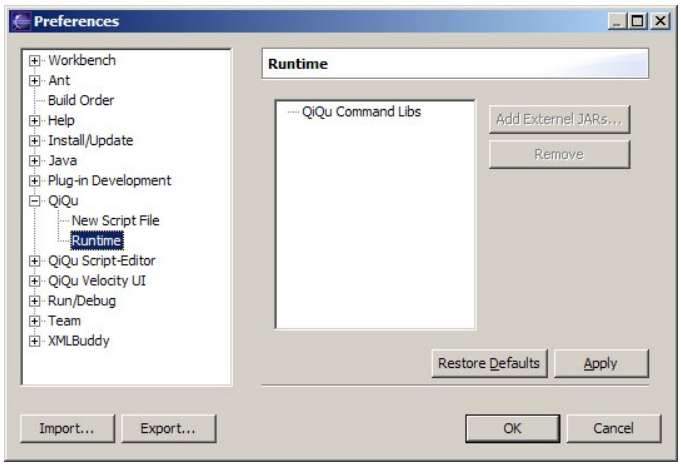
We will do that with an simple ant script that jars the classes-directory into a jar-file.

```
<project name="BuildDemoLib" default="buildlib" basedir=".">
        <property name="dist" location="dist" />
        <property name="classes" location="bin/classes" />
        <property name="filename" value="firstqiqulib.jar" />

        <target name="buildlib">
                <!-- Create the distribution directory -->
                <mkdir dir="${dist}" />

                <!-- Put everything in ${classes} into the jar file -->
                <jar jarfile="${dist}/${filename}" basedir="${classes}" />
        </target>
</project>
```

The library jar will be created in the "dist" directory. Now let's ...

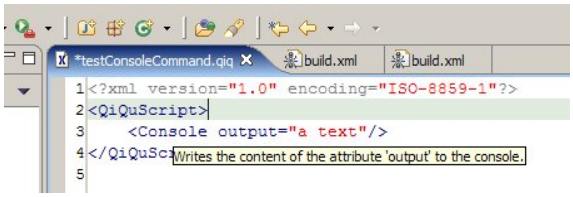
Register the library-jar to the QiQu-script editor

This can be done under the Eclipse-Preferences. Open them, select open the "QiQu"-tree and select runtime. You will see the following dialog:



In the "Runtime" dialog, select "QiQu Command Libs" and press the "Add External JARs..." button. Select the library file and press close the "Preferences" dialog by pressing the "OK" button.

Open the test-script in the "script" directory, make a small change and watch the error marker disappear. Moving the mouse cursor over the command tag will now show the description of the command.



Since the library is now registered inside the QiQu-system, you can launch the script directly: Open the context menu of the file in the navigator or package explorer and select "Run QiQu Script" under the menu group "QiQu".

Types of Command Parameter

As mentioned earlier in the tutorial, there are several different types which can be assigned to a command parameter. Each type has a special behaviour and for each type exists an appropriate class which represents the type. The BaseCommand class provides an appropriate factory method for each parameter type. The different parameter types are described in the **QiQu Commands Overview**, and the following table gives an overview of the explicit parameter type classes and their factory method.

Type	Class	Factorymethod [in BaseCommand
text	ch.aloba.qiqu.script.base.cmdparameter.CmdParamText	BaseCommand.addTextParam
nodeReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamNodeRead	BaseCommand.addNodeRead
saveNodeReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamNodeSave	BaseCommand.addNodeSave
documentReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamDocRead	BaseCommand.addDocReadP
saveDocumentReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamDocSave	BaseCommand.addDocSaveP
elementReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamEleRead	BaseCommand.addEleReadPa
saveElementReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamEleSave	BaseCommand.addEleSavePa
saveReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamRefSave	BaseCommand.addRefSavePa
readReference	ch.aloba.qiqu.script.base.cmdparameter.CmdParamRefRead	BaseCommand.addRefReadPa
readFile	ch.aloba.qiqu.script.base.cmdparameter.CmdParamFileRead	BaseCommand.addFileReadP
saveFile	ch.aloba.qiqu.script.base.cmdparameter.CmdParamFileSave	BaseCommand.addFileSavePa
valueexpression	ch.aloba.qiqu.script.base.cmdparameter.CmdParamValueExpr	BaseCommand.addValueExpr
xpath	ch.aloba.qiqu.script.base.cmdparameter.CmdParamXPath	BaseCommand.addXpathPara

Consult the javadoc of the parameter type classes to find out the methods provided. Also have a look at the core commands in order to see how the parameter type classes are used.

Write your first QiQu-Function

Inside a valueexpression parameter of QiQu-Commands, QiQu-Functions can be used. A QiQu-Function is a string manipulating action, which can take any number of parameters and which always returns a string. A userdefined QiQu-Function has to be derived from the class BaseFunction.

Again, we will make a simple example. Let's create a "Greeting"-function. The function will take two parameters. The first parameter defines the sex of a person ('m' or 'f') and the second parameter will contain the name of the person. Dependending on the sex of the person, the function will create the string "Hello Mr. *name*" or "Hello Ms. *name*".

The first step is to create a new java class, which we call "Greeting" and which is derived from the class "BaseFunction". Your class should look like the one shown next:

```
/*
 * Created on 20.01.2006
 */
package ch.aloba.qiqu.extend;

import ch.aloba.qiqu.script.base.BaseFunction;

/**
 * A simple QiQu-Function.
 *
 * @author Hansjoerg
 * @version $Revision: 1.7 $
 */
public class Greeting extends BaseFunction
{
}
```

Now, we have to implement the required methods.

The methods "getName", "getDescription" and "getFunctionGroup" are similar to the ones needed for a command. Therefore, we are going to add the following lines:

```
/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#getFunctionName()
 */
public String getFunctionName()
{
    return "greeting";
}

/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#getFunctionGroup()
 */
public String getFunctionGroup()
{
    return "text";
}

/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#getDescription()
 */
public String getDescription()
{
    return "Creates a greeting string depending on the sex and name of a person.";
}
```

Let's examine the rest of the needed methods in detail:

getReturnType

As mentioned, every QiQu-Function returns a String. However, QiQu-knows boolean function, which do return either the string "true" or "false". Boolean QiQu-Function should therefore return in this method the constant IBaseFuntion.TYPE_BOOLEAN. All other have to return IBaseFuntion.TYPE_TEXT.

getNumberOfParameters

This defines how many parameter the function has. Our "greeting"-function takes two parameters, therfore this method will return 2.

isLastParameterMultiple

This method returns true, if the last parameter can be repeated any number of times. Typical examples are the core boolean function like "and" and "or". They need at least two parameters, however, there can be any number of parameters. Therefore, this functions have to return true in this method, whereas our "greeting" method will return false.

setParameters

Inside this method, you have to call for every parameter the method "addParameter". The first parameter of this function is the index of the parameter as string, the second is the description of the parameter.

Implementing the above mentioned methods will add the follwing code to our class.

```
/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#getReturnType()
 */
public int getReturnType()
{
    return IBaseFunction.TYPE_TEXT;
}

/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#getNumberOfParameters()
 */
public int getNumberOfParameters()
{
    return 2;
}

/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#isLastParameterMultiple()
 */
public boolean isLastParameterMultiple()
{
    return false;
}

/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#setParameters()
 */
protected void setParameters()
{
    addParameter("1", "The sex of th person, either 'm' or 'f'");
    addParameter("2", "The name of the person.");
```

```
}
```

evaluateFunction

The last method to be implemented is "evaluateFunction". This Method will contain the logic of our function.

The method receives 2 parameters. The first one is a String[] (params) with the evaluated values of the QiQu-Function. The second is an Expression[] (expressions) with the Abstract-Syntax-Trees of the parametes. Calling the getValue-method of an Expression will return the evaluated value.Therfore, `params[i].equals(expressions[i].getValue())`. To implement our function, we just have to consider the evaluated params.

```
/**
 * @see ch.aloba.qiqu.script.base.BaseFunction#evaluateFunction(java.lang.String[], ch.alob
 */
public String evaluateFunction(String[] params, Expression[] expressions) throws Exception
{
    StringBuffer greetStr = new StringBuffer("Hello ");
    if (params[0].equals("m"))
    {
        greetStr.append("Mr.");
    }
    else if (params[0].equals("f"))
    {
        greetStr.append("Ms.");
    }
    greetStr.append(" ").append(params[1]);
    return greetStr.toString();
}
```

Test and debug your QiQu-Function

In order to register the function inside the QiQu-System, we need to add an appropriate entry in the QiQu-library descriptor:

```
<QiQuLib>
        <function fqn="ch.aloba.qiqu.extend.Console"/>
</QiQuLib>
```

Now let's add the following line to our testscript:

```
<EchoText InfoText="greeting('m','Sears')"/>
```

Now launch the testclass and in the console output, the entry

```
[INFO] EchoText - Hello Mr. Sears
```

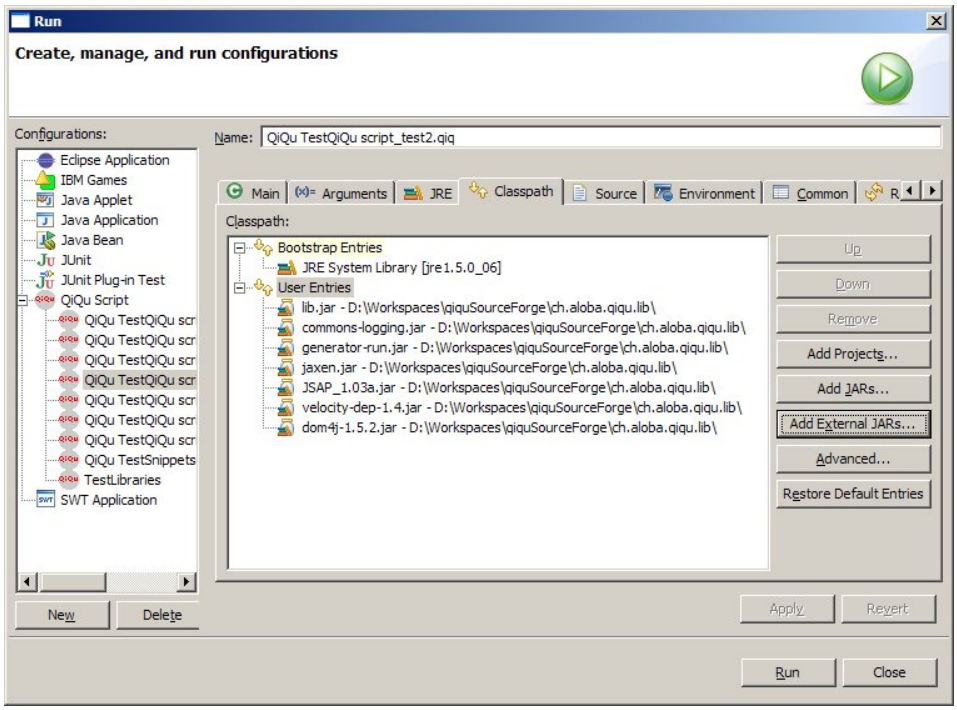
will appear.

In order to make the function known to the editor, you have to create a new jar, and register it freshly in the preferences of QiQu.

Adding additional libraries to the classpath

When you develop additional commands and functions, it can be necessary to use additional libraries. If you launch a qiquscript from the command line, you simple need to add the additional libraries to your classpath.

When you run your script inside eclipse, you need to add the additional libraries in the classpath register of your launch configuration:



A default launch configuration is created the first time, you launch your script with the Run or Debug action from the contextmenu. It's named "QiQu [projectname] [filename]". You can select your additional libraries in this configuration. The next time you launch the script by selecting Run or Debug from the context menu, the additiona libraries will be loaded.