IBM Application Runtime Expert for i

Table of Contents

Writing Custom Plugins

The IBM Application Runtime Expert for i (ARE) architecture provides a simple, yet powerful, mechanism for augmenting the verification that is defined in a deployment template. This mechanism is referred to as custom plugins.

All verification in the ARE is performed by plugins. Verification of authority, file attributes, PTFs, etc is done by plugins that come pre-packaged with the ARE. But in reality, these pre-packaged plugins are no different than custom plugins; they all use the exact same plugin architecture that custom plugins do!

The ARE is implemented in Java; therefore, in the ARE environment, a plugin is simply a Java class that extends the `com.ibm.are.plugin.BasePlugin` class. The BasePlugin class provides the framework necessary to rapidly develop plugins that integrate seamlessly into the ARE environment. The BasePlugin class provides the following set of implemented features available for use by any sub-class:

- A fully initialized, ready to use logger
- Reporting of status and problems
- Ability to specify how to fix detected problems

Writing a plugin is simple; so simple in fact that we're going to start our discussion of how to write a custom plugin with the complete source code for a plugin . The plugin below simply reports the read and write access attributes of a file. Note that ARE already has a plugin that implements similar functionalities, so the plugin below is for illustrative purposes only.

```
/*
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2010 All Rights Reserved
 *
 * The source code for this program is not published or other-
 * wise divested of its trade secrets, irrespective of what has
 * been deposited with the U.S. Copyright Office.
 *
 * DISCLAIMER:
 * This program contains code made available by IBM Corporation on an
 * "AS-IS" basis. This sample code has not been thoroughly tested under all conditions.
 * IBM, therefore, cannot guarantee or imply reliability, serviceability, or function.
 * IBM provides no program services for this material.
 *
 * THIS MATERIAL IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR
 * IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE
 * EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU.
 * IN NO EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER
 * CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS MATERIAL INCLUDING, WITHOUT LIMITATION, ANY
 * LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION
 * HANDLING SYSTEM OR OTHERWISE, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * You may copy and modify this source code and may redistribute modified versions of this
 * source code, provided however that you do not alter or delete any copyright information
 * or notices contained in the source code.
 *
 */
package com.ibm.are.example;

import com.ibm.are.common.Version;
import com.ibm.are.plugin.BasePlugin;
import com.ibm.are.plugin.ResultInfo;
import java.io.File;


public class DemoFileChecker extends BasePlugin {
```

```
        protected Version initPluginVersion() {
                return new Version(1, 0, 0);
        }

        public String getCommonName() {
                return "Demo File Checker";
        }

        public String getDescription() {
                return "Reports the read and write access attributes for a file";
        }

        protected ResultInfo runImpl() {
                // Do actual checking here
                File fileToCheck = new File("/tmp/plugintest.txt");
                reportStep("Printing attributes for file " + fileToCheck);
                reportStepDetail ("Read access: " + fileToCheck.canRead());
                reportStepDetail ("Write access : " + fileToCheck.canWrite());
                return new ResultInfo(true);
        }
}
```

As you can see from this sample plugin, there is very little “boilerplate” work required when creating a plugin. The four methods included in the DemoFileChecker plugin are required by any plugin; the purpose of each of these methods is discussed below.

Details About Methods to Implement

The BasePlugin class is an abstract base class. When you extend this class, you are required to implement four methods. Implementation of these four methods is all that is required to have a plugin capable of running in the ARE environment. Here is a summary and description of each method:

initPluginVersion - Every plugin has a version associated with it. This version is represented by the com.ibm.are.common.Version class. The BasePlugin super class contains a protected field where the Version is stored. This field is initialized during the BasePlugin's constructor by invoking the initPluginVersion method. So all you need to do to implement this method is return a newly created Version object that is initialized with the current version of your plugin. Typically, a brand new plugin will be assigned a version of 1.0.0.

getCommonName - Every plugin has a common name associated with it. Think of a common name as a descriptive, human-readable name for your plugin. You are allowed to use characters such as spaces for your common name. The DemoFileChecker plugin has a common name of ‘Demo File Checker’. There is no requirement that the common name have any relation to the class name, but if it makes sense it is certainly permissible and, in many cases, it makes a lot of sense.

getDescription - Every plugin has a description associated with it. The description should provide a detailed explanation of what the plugin's purpose is. The DemoFileChecker plugin is a very simple plugin that reports the read and write access attributes of a file, which is exactly what its description says it does.

runImpl – Last, but certainly not least, is the runImpl method. This is the most important method, since it is where you implement the functionality of your plugin. This class requires a return object of type `com.ibm.are.plugin.ResultInfo`. This return object is meant to communicate any result information from the plugin back to the ARE runtime itself. In most cases, plugins need only to communicate a final result. By convention, if a plugin completes running, regardless if any problems were found, a ResultInfo with a value of 'true' is returned.

Using the Reporting Framework

Reporting is one of the most basic functions a plugin performs. Reporting refers to two things: reporting the status of verification and reporting any problems detected while verifying. Reporting verification status is done using the following methods:

- reportStep
- reportStepDetail
- reportSubStep
- reportSubStepDetail

These methods provide a way to report status in a hierarchical manner. Let's take a look at some sample code to provide a more concrete example. Note that the code below would be part of the plugin's implementation, and therefore would be located in the runImpl() method.

```
File fileToCheck = new File("/tmp/plugintest.txt");
reportStep("Printing attributes for file " + fileToCheck);
reportStepDetail("Access Permissions");
reportSubStep("Read access: " + fileToCheck.canRead());
reportSubStep("Write access : " + fileToCheck.canWrite());
reportStepDetail("Other Attributes");
reportSubStep("Size: " + fileToCheck.length());
reportSubStep("Last Modified: " + fileToCheck.lastModified());
reportSubStep("Is hidden: " + fileToCheck.isHidden());
```

When the above code is run, it produces output in the ARE report that looks like this:

> Printing attributes for file /tmp/plugintest.txt
  o Access Permissions
    - Read access: true
    - Write access : true
  o Other Attributes

- Size: 15
- Last Modified: 1266784893000
- Is hidden: false

In the above example, all of the code is reporting status; there is no code that reports any problems. When a problem is detected by a plugin, it should be reported at the appropriate severity level. The problem severity can be one of three levels: Error, Warning, Info. The purpose of different severity levels is to give plugin authors some latitude in how a problem gets reported. For example, if a plugin is checking for the existence of an application configuration file, and that file does not exist, this may be a critical problem. In this case, the problem can be reported with a severity level of Error. However, if the configuration file is optional, then the fact that it does not exist is not nearly as bad, but it may be noteworthy and thus should be reported. In this case, the plugin can report the missing file using the Warning or Info problem severity level.

To report a problem, any of the following methods can be used:

- **reportError** – report a problem with the highest severity (Error)
- **reportWarning** – report a problem with medium severity (Warning)
- **reportInfo** – report a problem with low severity (Info)

So let's take a look at a slightly modified version of the reporting example code. This example will report a problem if the file's 'hidden' attribute is not set:

```
File fileToCheck = new File("/tmp/plugintest.txt");
reportStep("Printing attributes for file " + fileToCheck);
reportStepDetail("Access Permissions");
reportSubStep("Read access: " + fileToCheck.canRead());
reportSubStep("Write access : " + fileToCheck.canWrite());
reportStepDetail("Other Attributes");
reportSubStep("Size: " + fileToCheck.length());
reportSubStep("Last Modified: " + fileToCheck.lastModified());

if (fileToCheck.isHidden()) {
        reportSubStep("File is hidden as expected");
}
else {
        reportError("File is NOT hidden, but it should be");
}
```

When the above code is run, it produces output in the ARE report that looks like this:

> Printing attributes for file /tmp/plugintest.txt
  o Access Permissions
    - Read access: true
    - Write access : true
  o Other Attributes
    - Size: 15
    - Last Modified: 1266784893000
ERROR! File is NOT hidden, but it should be

As you can see, finding problems in the output is very simple just by looking for text the is aligned with the leftmost column, or by searching for one of the problem keywords (ERROR!, Warning, Info). An even simpler way to find any problems reported by a plugin is to look at the summary report that is produced when the ARE verifies a deployment.

The code example above is very simple in that it only provides a single line of text when reporting the problem. What if you want to provide some additional details about a problem when you are reporting it? Luckily, ARE provides a mechanism for doing exactly this. Let's take a look at a slightly modified version of the previous code example:

```
File fileToCheck = new File("/tmp/plugintest.txt");
reportStep("Printing attributes for file " + fileToCheck);
reportStepDetail("Access Permissions");
reportSubStep("Read access: " + fileToCheck.canRead());
reportSubStep("Write access : " + fileToCheck.canWrite());
reportStepDetail("Other Attributes");
reportSubStep("Size: " + fileToCheck.length());
reportSubStep("Last Modified: " + fileToCheck.lastModified());

if (fileToCheck.isHidden()) {
        reportSubStep("File is hidden as expected");
}
else {
        ProblemContext probCtx = reportError("File is NOT hidden, but it should be");
        probCtx.details("Here is more information about the problem");
        probCtx.details("And here is even more information about the problem");
}
```

When the above code is run, it produces output in the ARE report that looks like this:

```
> Printing attributes for file /tmp/plugintest.txt
    o Access Permissions
        - Read access: true
        - Write access : true
    o Other Attributes
        - Size: 15
        - Last Modified: 1266784893000
ERROR! File is NOT hidden, but it should be
Details: Here is more information about the problem
Details: And here is even more information about the problem
```

The ProblemContext object is an instance of the `com.ibm.are.report.ProblemContext` class. The sole purpose of this object is to provide a plugin with a way to report additional details about a problem that is reported using the `reportError`, `reportWarning`, or `reportInfo` methods. So why use this problem context object? Why not just report the details using the same method that

was used to report the problem itself? The answer lies in how ARE counts the number of problems found by a plugin. Each time a problem is reported, the count of the number of problems is incremented. But any details associated with a problem are not counted. So in the example code above, the `reportError` would cause the problem count to go up by one, but the two subsequent `details` for that problem do not have any affect on the problem count. If we had simply used `reportError` three times; once to report the problem and twice more to report its details, the resulting problem count would have been (incorrectly) incremented three times instead of just once.

One final, very important, thing to remember about reporting is that all status and problems reported by custom plugins are pulled into the reports (detailed, summary, XML) that are generated by ARE when verifying a system.

Compiling Your Plugin

You can use the standard Java compiler tools to compile your plugin. Because you are extending and using classes that are part of the ARE core, you will need to include the ARE core jar in your classpath when you compile. The ARE core jar can be found here:

/QIBM/ProdData/OS/OSGi/healthcheck/lib/arecore.jar.

The ARE core jar is available on any IBM i system with the following Group PTF levels:
IBM i 5.4 – HTTP Group PTF SF99114 level 23 or later
IBM i 6.1 – HTTP Group PTF SF99115 level 14 or later
IBM i 7.1 – HTTP Group PTF SF99368 level 2 or later

If your plugin is using the IBM Toolbox for Java, you will also need to add that jar to your classpath. That jar can be found here:

/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar

Note that both of the aforementioned jar files are already part of the ARE's runtime classpath, so you do not need to do anything to have access to the classes in these jars at runtime.

To compile the source code of the sample plugin DemoFileChecker.java, follow these steps:

1) Go into a temporary directory somewhere in IFS. For this example, we will use /tmp/plugin as the temporary directory.
2) In /tmp, plugin, run this command: mkdir -p com/ibm/are/example
3) Put DemoFileChecker.java into the /tmp/plugin/com/ibm/are/example directory.
4) Set your Java home:
    a. IBM i 5.4: export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit

b. IBM i 6.1: export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
c. IBM i 7.1: export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit

5) Compile the code: javac -classpath .:/QIBM/ProdData/OS/OSGi/healthcheck/lib/arecore.jar:/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar com/ibm/are/example/DemoFileChecker.java
6) Once the source is compiled, it can be packaged into a jar with this command: jar cvf DemoFileChecker.jar com/ibm/are/example/DemoFileChecker.class

IMPORTANT: Once you have packaged DemoFileChecker.class into the jar, you need to delete the DemoFileChecker.class file. This will avoid problems where the ARE Core finds the DemoFileChecker.class in IFS rather than in the jar file. To remove the DemoFileChecker.class file, run this command:

rm com/ibm/are/example/DemoFileChecker.class

Testing Your Plugin

The ARE allows you to add custom plugins to your deployment template. However, there is a way to run an individual plugin without packaging it into a deployment template. This is very useful when you are developing a new plugin and want to test it without going through the process of adding it to your deployment template and then re-building the template.

To run a plugin without packaging it into a deployment template, you need to first put the plugin into a jar file. A jar file is the standard packaging mechanism for anything in the ARE environment, whether it is an individual plugin or an entire deployment template. Once the plugin is in a jar file (the steps in the previous section instruct you how to put your compiled plugin into a jar) you can use the arePlugin.sh script to run it. Here is the syntax:

/QIBM/ProdData/OS/OSGi/healthcheck/bin/arePlugin.sh -pluginName *plugin_class_name* -elementPath *plugin_jar_file*

IMPORTANT: The plugin class name must be its fully qualified package and class name. For example, the DemoFileChecker plugin's full package and class name is com.ibm.are.example.DemoFileChecker.

To run the DemoFileChecker plugin, you would use the following command line:

/QIBM/ProdData/OS/OSGi/healthcheck/bin/arePlugin.sh –pluginName **com.ibm.are.example.**DemoFileChecker –elementPath DemoFileChecker.jar

Note that the command above assumes that the DemoFileChecker.jar is in the current working directory. If the jar file is in a different directory, then the jar name needs to be fully qualified.

Building a Custom Plugin Using Rational Application Developer

There are many different Java development environments that can be used to write ARE custom plugins. In this section, we will walk through creating a custom plugin using the Rational Application Developer (RAD) environment. The example presented in this section provides a step by step guide, with accompanying screenshots, for creating a custom plugin and exporting it to a Java JAR file.

1. Create Java Project

The first step is to create a new Java project using the **File -> New -> Java Project** menu option.

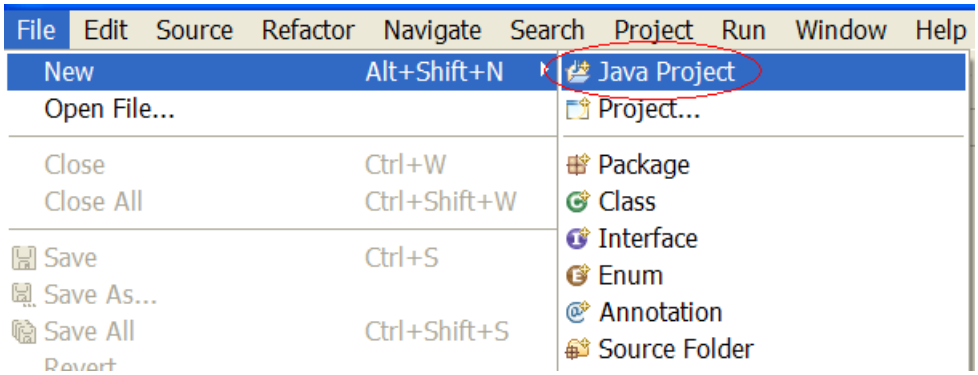

Figure 1 – Selecting new Java project menu option

Input the project name; for this example we use MyFirstPlugin as the project name. Note that the project must use JDK 1.4.2 for its Java compiler. In the example below, we are using a JDK 5 compiler.

Important: If you select a JDK 6 compiler, your plugin class will be compiled to target a JDK 6 JVM, which means a JDK 6 JVM will be required on any IBM i system where your plugin is used for verification. Because JDK 6 may not be installed on older IBM i systems, it is recommended to use either a JDK 1.4.2 or JDK 5 compiler so that your plugin is capable of running on the widest possible set of IBM i systems.

Leave the other fields as default and click the Finish button.

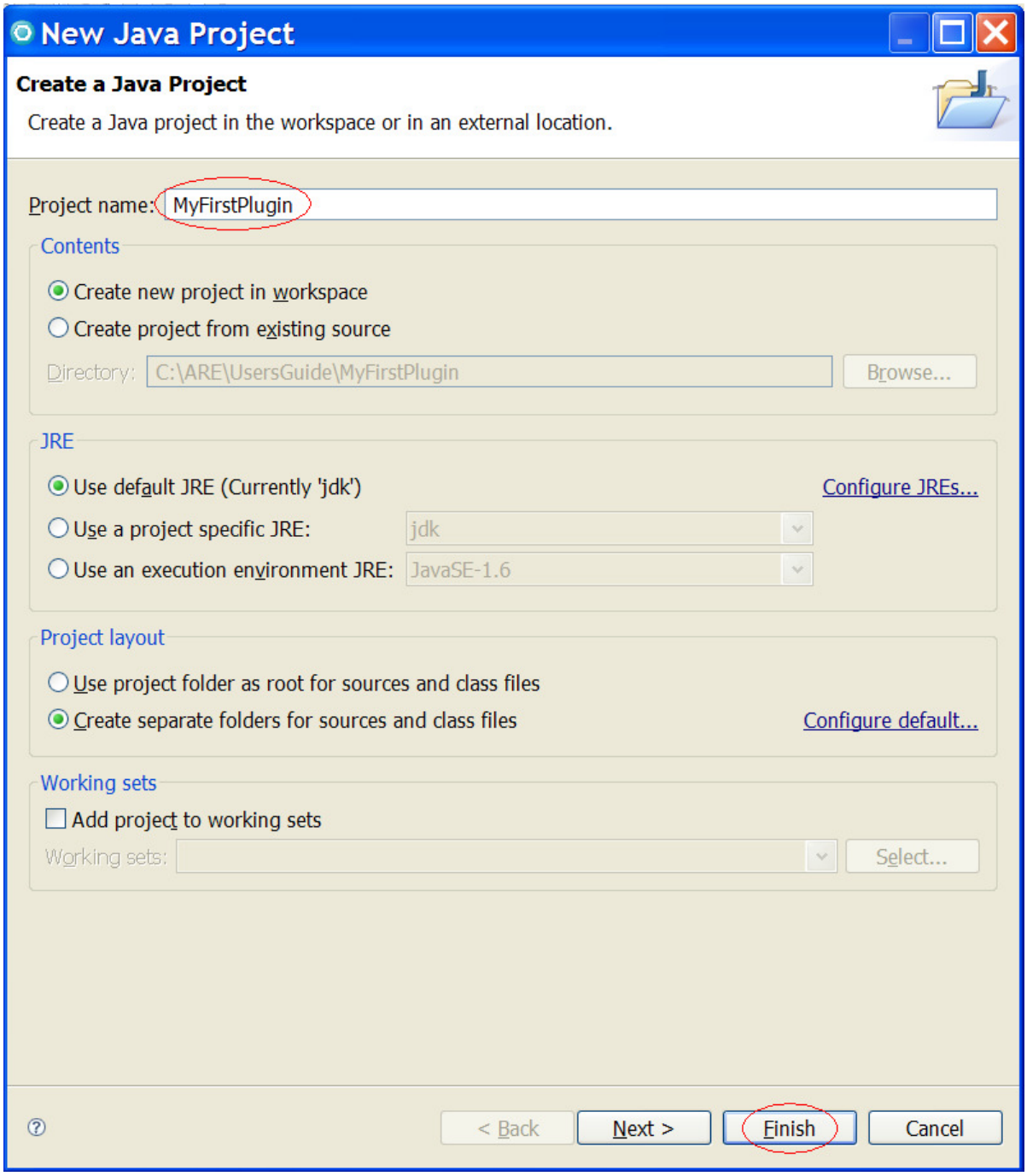

Figure 2 – Creating a new Java project

The project will create with no further actions necessary. Your perspective should be automatically changed to the Java perspective, but if it is not, you can switch to the Java perspective by clicking the “Open Perspective” button in the upper right corner of the window.

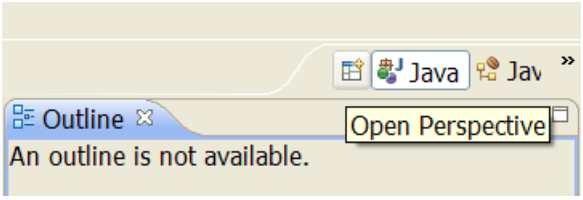

Figure 3 – Open perspective button

2. Copy arecore.jar file to local system

Compiling an ARE plugin requires, at a minimum, the *arecore.jar* jar to be included in the build path. The ARE core jar can be found on any IBM i system, 5.4 or newer, with the appropriate HTTP Group PTF level installed. See the Compiling Your Plugin section for the required HTTP Group PTF levels in order for you to have the ARE core jar file on your IBM i system.

The ARE core jar can be found on an IBM i system in the following location:
/QIBM/ProdData/OS/OSGi/healthcheck/lib/arecore.jar

Copy this file to local system where you are using RAD to build your plugin.

If your plugin makes use of the IBM Toolbox for Java, you will also need to add that jar file to your project's build path. The Toolbox jar file can be found on an IBM i system in the following location:
/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar

3. Add arecore.jar to the Java project

Now that you have the necessary jar files copied onto your system where you are using RAD to create your plugin, you need to add the arecore.jar to the project build path. In the Package Explorer, find the project you just created, right click on the project name, and on the popup menu, select **Properties**.

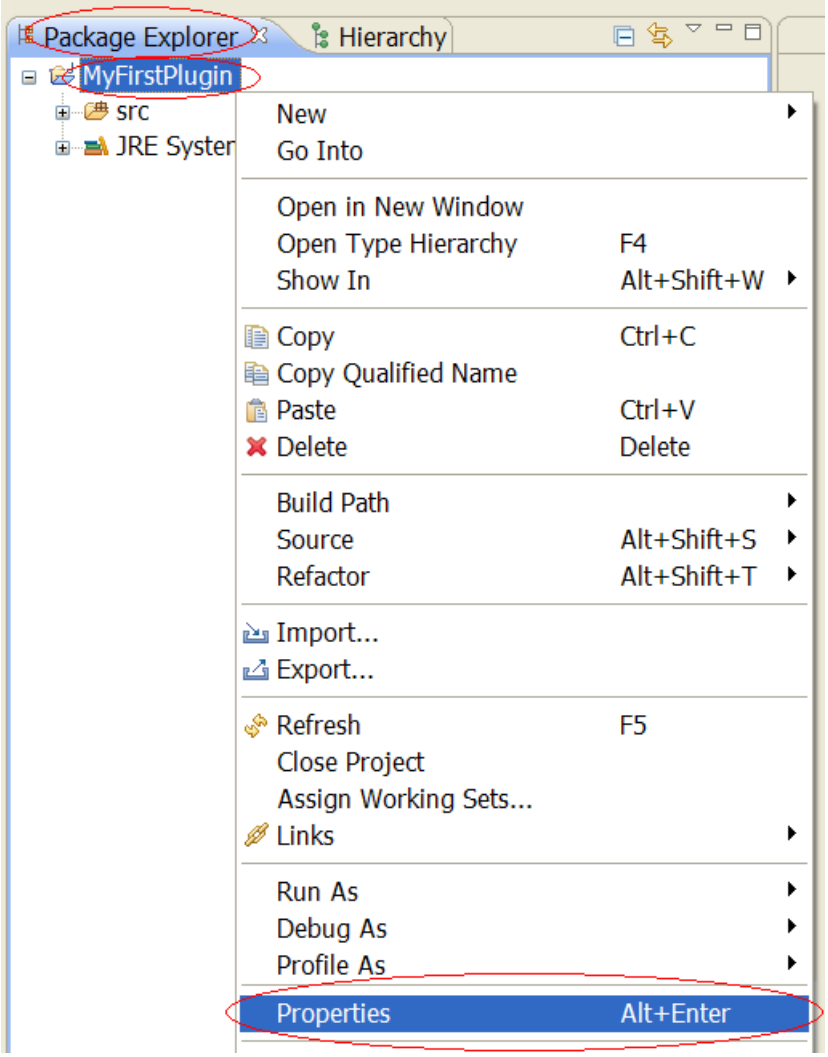

Figure 4 – Project properties

4. Add arecore.jar to the Java Build Path

On the properties dialog, select **Java Build Path** in the left pane, select the **Libraries** tab, and click the **Add External JARs** button.

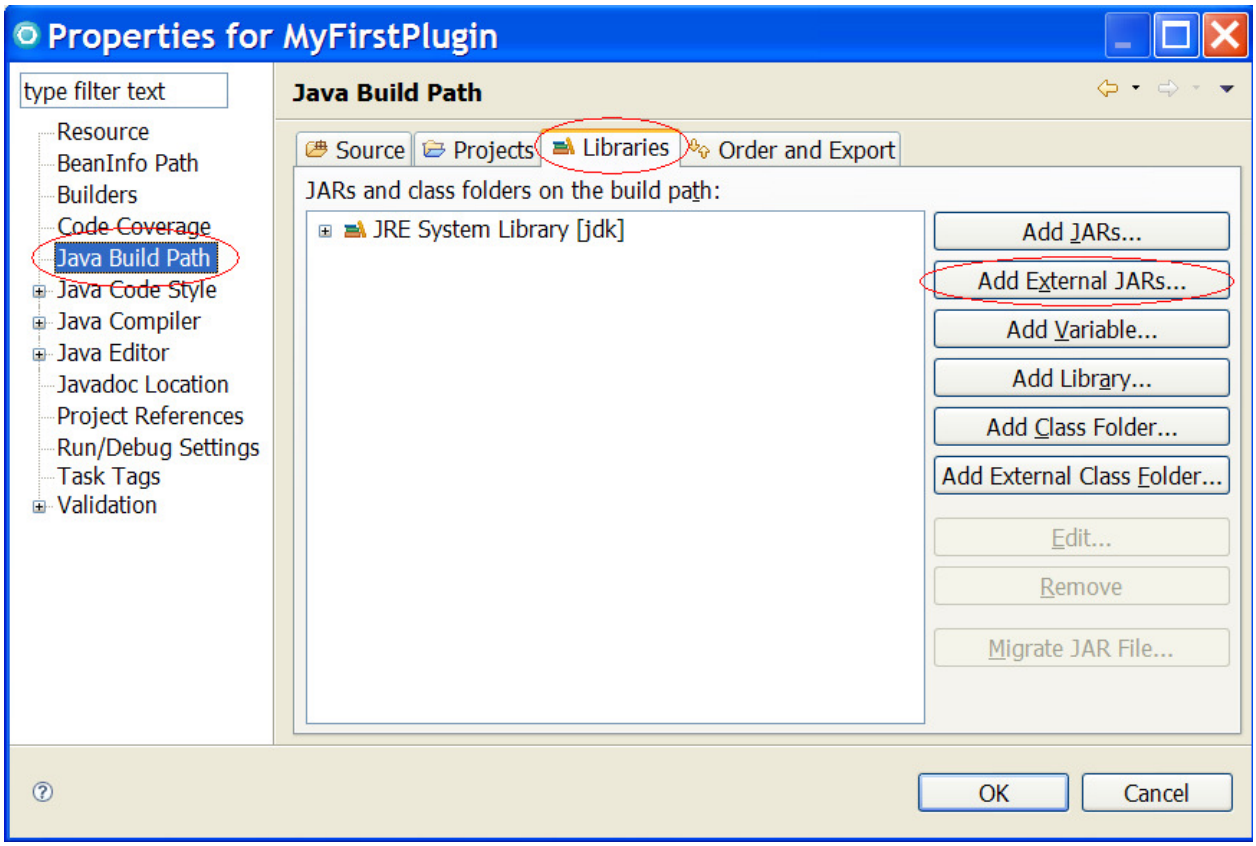

Figure 5 – Configure Java build path

Browse your local file system and locate the *arecore.jar*, select it and click **Open**.

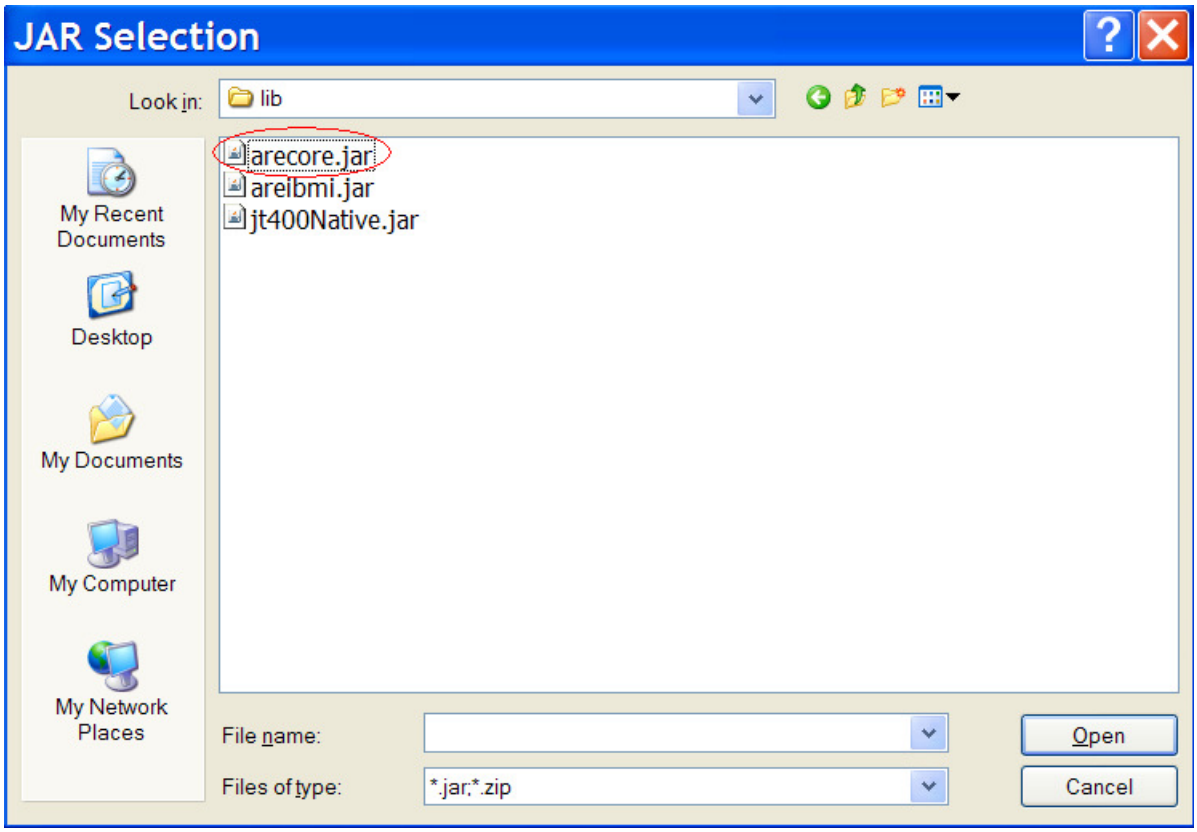

Figure 6 – Select jar to add to Java build path

Now arecore.jar has been added to the build path, as shown in the following figure 7 below. Click the OK button to finish, or repeat this process to add other jar files that you need to your build path.

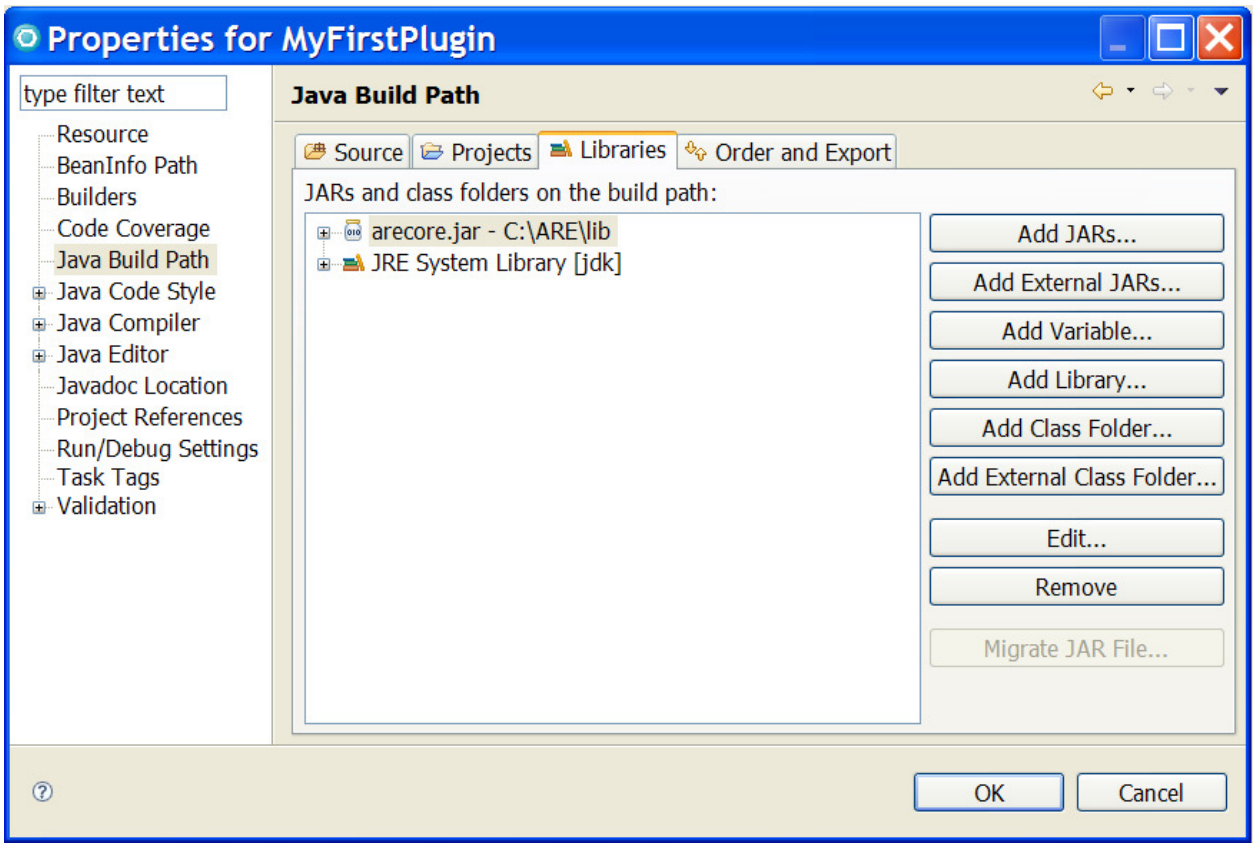

Figure 7 – ARE core jar added to the Java build path

5. Create new ARE plugin

The build path for your project has now been configured, which means you are ready to create a new ARE plugin. In the Package Explorer, right click on 'src' folder and select **New -> Class** from the popup menu.

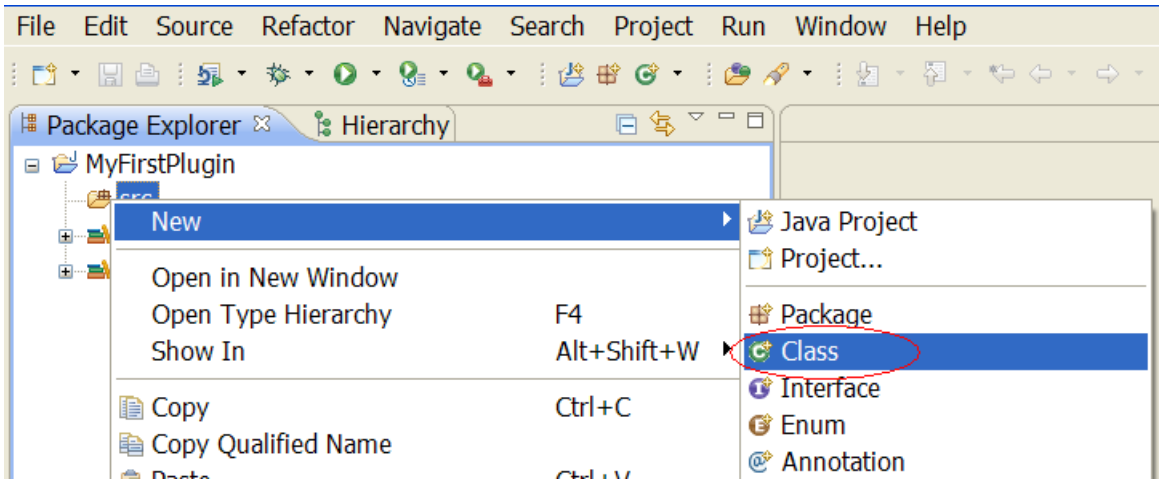

Figure 8 – Add a new class to the project

You must provide a name for the class. In this example, we use the name `com.ibm.are.example.MyFirstPlugin`, where 'com.ibm.are.example' is the package name, and 'MyFirstPlugin' is the class name. Set the **Superclass** for the plugin to **`com.ibm.are.plugin.BasePlugin`**.

IMPORTANT: Make sure that the 'Inherit abstract methods' selection box is checked. This will ensure that the stubs for all required methods will be added to your newly created plugin class.

Click the Finish button.

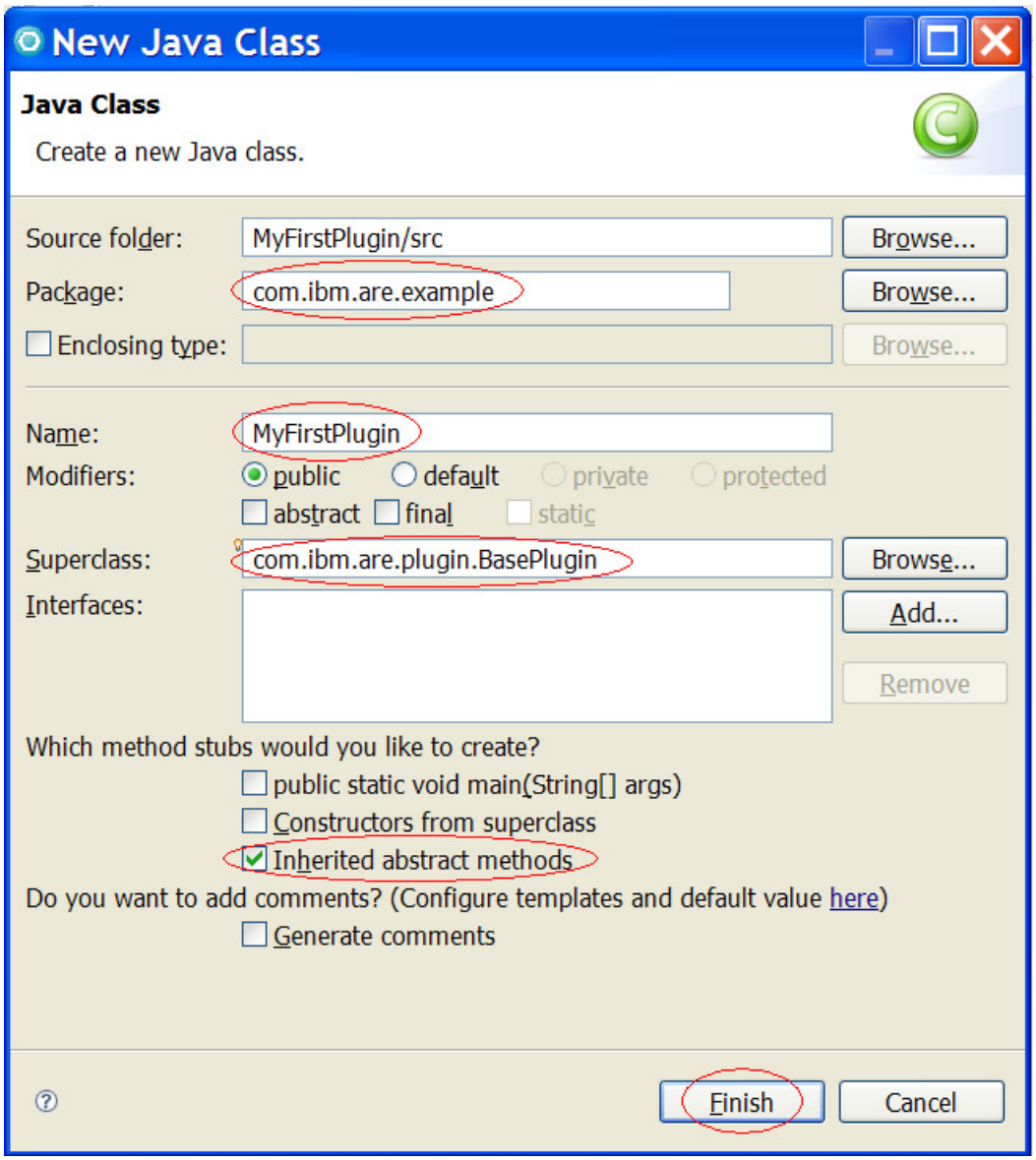

Figure 9 – Specify details for new class

The MyFirstPlugin class (plugin) will be automatically created, and stubs for all of the required methods for implementing an ARE plugin will be created (see figure 10 below):

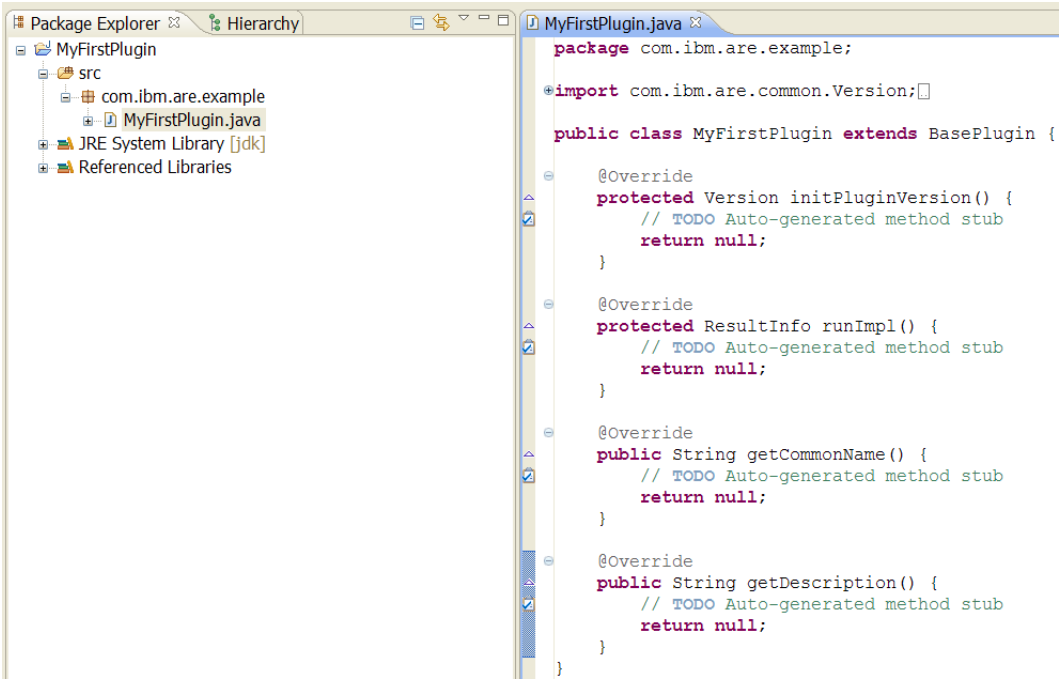

Figure 10 – Source for the newly created class

If you did not check the 'Inherit abstract methods' selection box when creating the plugin class, your new plugin will show an error/problem similar to figure 11:

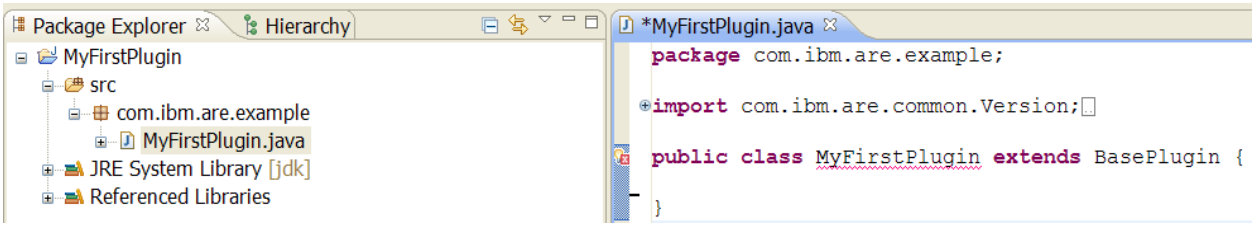

Figure 11 – Empty class without inherited abstract methods

If your project is missing the required method stubs, fixing this is simple. To fix this problem, and get the necessary method stubs inserted into your class, click the "x" icon at left of the editor, which displays a menu of options for fixing the problem. Click on the **Add unimplemented methods** option, and stubs for the required plugin methods will be added to your class.

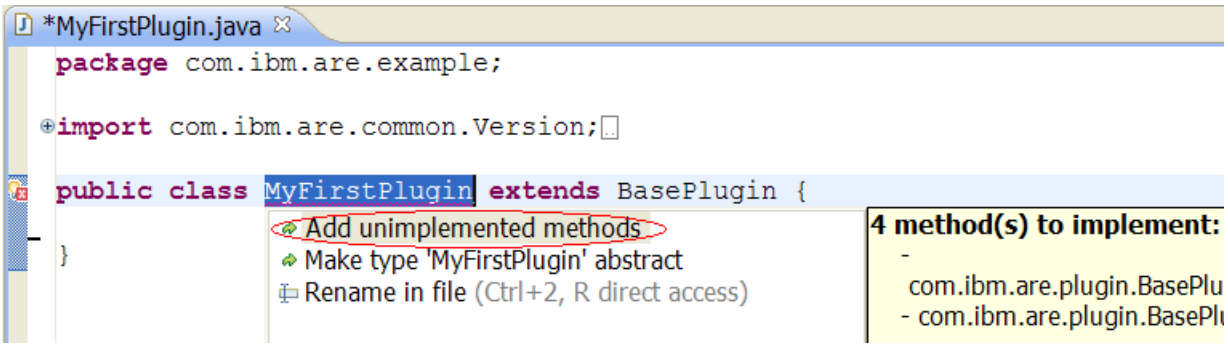

Figure 12 – Fixing the problem by adding the unimplemented methods

6. Method Stub Implementation

To complete this example, we need to implement the four method stubs. These method stubs provide the plugin with a version, name, description, and, most importantly, an implementation in the `runImpl` method. For the purposes of this example, the implementation is simply to report a status message of "Hello World!".

```
/*
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2010 All Rights Reserved
 *
 * The source code for this program is not published or other-
 * wise divested of its trade secrets, irrespective of what has
 * been deposited with the U.S. Copyright Office.
 *
 * DISCLAIMER:
 * This program contains code made available by IBM Corporation on an
 * "AS-IS" basis. This sample code has not been thoroughly tested under all conditions.
 * IBM, therefore, cannot guarantee or imply reliability, serviceability, or function.
 * IBM provides no program services for this material.
 *
 * THIS MATERIAL IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR
 * IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE
 * EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU.
 * IN NO EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER
 * CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS MATERIAL INCLUDING, WITHOUT LIMITATION, ANY
 * LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION
 * HANDLING SYSTEM OR OTHERWISE, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * You may copy and modify this source code and may redistribute modified versions of this
 * source code, provided however that you do not alter or delete any copyright information
 * or notices contained in the source code.
 *
 */
package com.ibm.are.example;

import com.ibm.are.common.Version;
import com.ibm.are.plugin.BasePlugin;
import com.ibm.are.plugin.ResultInfo;

public class MyFirstPlugin extends BasePlugin {

	protected Version initPluginVersion() {
		return new Version(1, 0, 0);
	}

	protected ResultInfo runImpl() {
		this.reportStep("Hello World!");
		return new ResultInfo(true);
	}
```

```
        public String getCommonName() {
                return "My First Plugin";
        }

        public String getDescription() {
                return "My very first plugin – a simple hello world example";
        }
}
```

7. Export the plugin

That's it! The plugin is now ready. To make use of the plugin in a template, or to test it as a standalone plugin, you need to export the plugin into a JAR file . Right click the project name, and click **Export** from the popup menu.

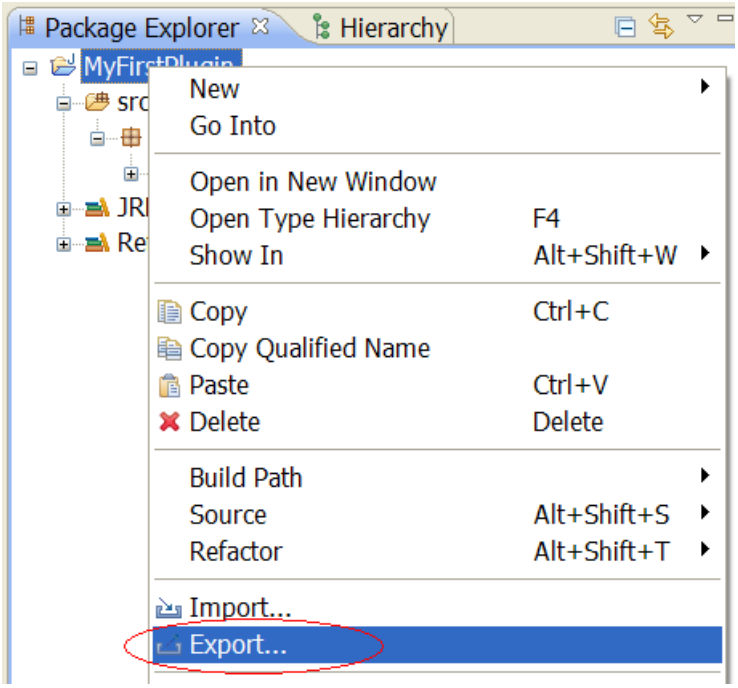

Figure 13 – Export the plugin

Select Java -> JAR file, and click Next.

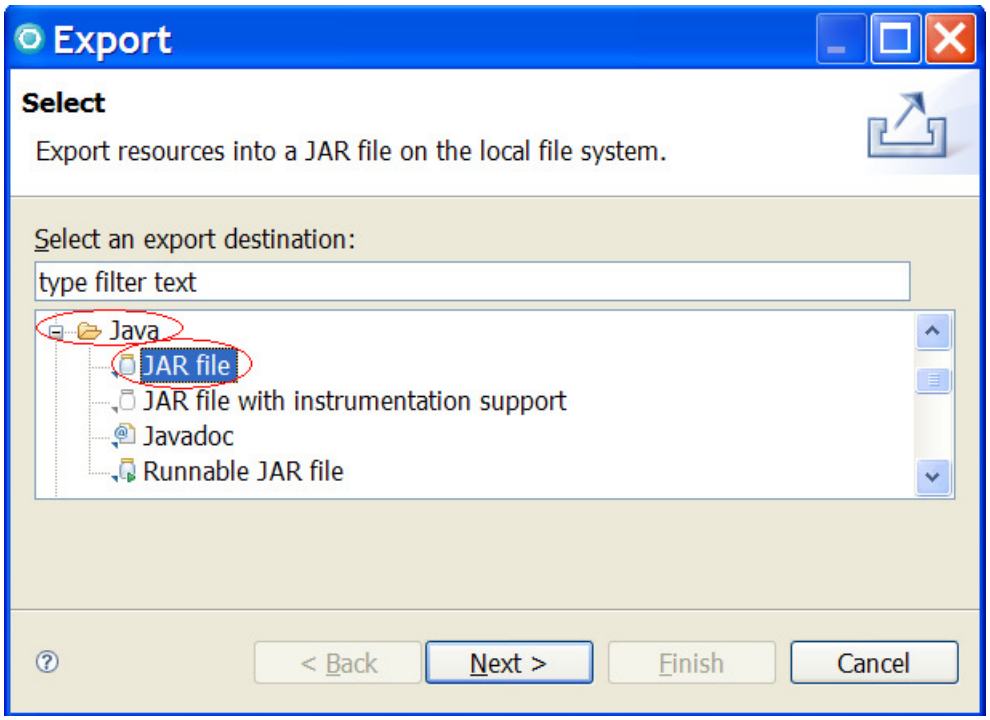

Figure 14 – Export the plugin to a Java JAR file

Set a destination for the JAR file. In this example, we are exporting the JAR to C:\ARE\MyFirstPlugin.jar. Once the destination for the JAR file is set, Click the Finish button.

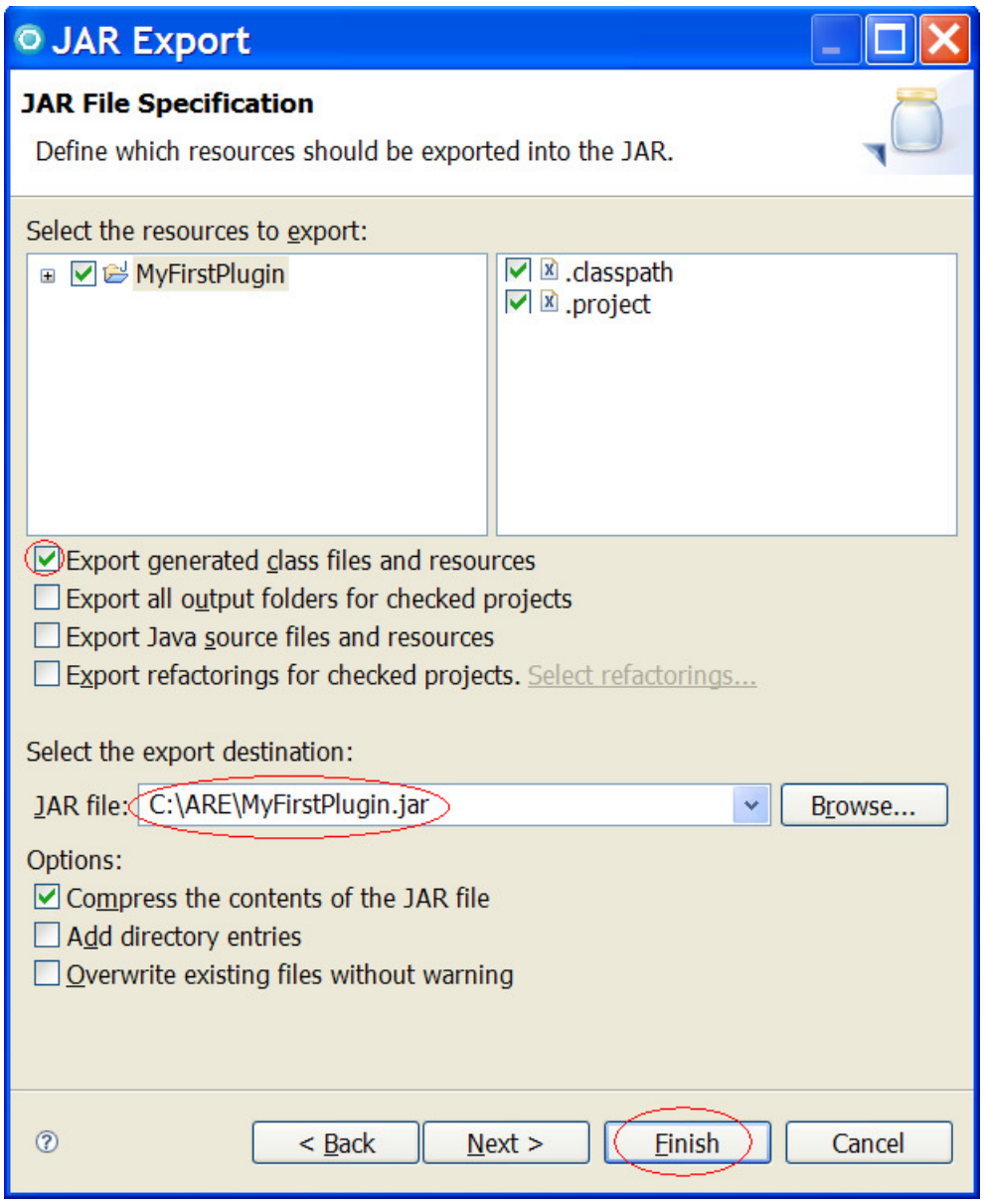

Figure 15 – Specify export items and options

8. Plugin Testing

To test this plugin, you need to first put the jar onto an IBM i system. For example, you could put the jar into the /tmp directory, then run the following command from QShell:

/QIBM/ProdData/OS/OSGi/healthcheck/bin/arePlugin.sh -elementPath /tmp/MyFirstPlugin.jar -pluginName com.ibm.are.example.MyFirstPlugin

You can also add your custom plugin to a new, or an existing, deployment template using the ARE Web user interface. For more information about testing your plugin, see

the Testing Your Plugin section of this document. For information about how to add a custom plugin to a template using the ARE Web interface, see the **Adding a Custom Plugin to a Template** document on the ARE product Web site:

http://www.ibm.com/systems/power/software/i/are/documentation.html

Advanced Plugin Topics

Logging

The `com.ibm.are.plugin.BasePlugin` class provides a ready to use logger. The logger is an instance of the `java.util.logging.Logger` class. All plugins share the same logger, with the output going into a file named are.log.0. Log files are aged, so subsequent runs of the ARE will result in the are.log.0 file being renamed to are.log.1 to make room for a new are.log.0 file. The maximum number of log files kept is five (are.log.0 through are.log.4).

The default logging level for the ARE is INFO. To change the log level, specify the property are.log.level=*desired_level* when starting the ARE using the runARE.sh script. If you are using the Console to invoke the ARE, you can specify the are.log.level property by clicking on the Runtime properties button. The log level can be set to any of the following values:

- SEVERE (minimum amount of logging)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (maxim mum amount of logging)

It is by no means required that you make use of the logger provided by the BasePlugin class. However, the logger provided by BasePlugin should suffice for most plugins, especially if all that is desired is some simple logging and an easy mechanism (the are.log.level property) to set the log level during ARE startup.

Providing Input to Plugins

Plugin can have input values passed to them via properties on the command line (or via the Runtime properties when using the Console). Use the following syntax to pass parameters to plugins via the command line:

-property *key*=*value*

Using the DemoFileChecker we previously created as an example, the file to be checked is hard coded in the plugin. Instead of hard coding the file name, let's pass the name of the file to check via the command line, enabling the plugin to check any desired file. To accomplish this, the command line to test the plugin would be:

/QIBM/ProdData/OS/OSGi/healthcheck/bin/arePlugin.sh –pluginName com.ibm.are.example.DemoFileChecker –elementPath DemoFileChecker.jar **–property myTargetFile=/tmp/plugintest.txt**

The -property argument will be processed by ARE, and it will create a property named *myTargetFile*, with a value */tmp/plugintest.txt*. To access this property from the plugin, use the `java.lang.System.getProperty` method. The updated plugin code to process the input argument looks like this:

```
protected ResultInfo runImpl() {
        // Do actual checking here
        String fileName = System.getProperty("myTargetFile");
        File fileToCheck = new File(fileName);
        reportStep("Printing attributes for file " + fileToCheck);
        reportStepDetail ("Read access: " + fileToCheck.canRead());
        reportStepDetail ("Write access : " + fileToCheck.canWrite());
        return new ResultInfo(true);
}
```

The *-property* parameter can be specified for the following ARE scripts. If more than one property needs to be specified, the -property parameter can be repeated multiple times.

runARE.sh
arePlugin.sh
areServices.sh
areFix.sh

Loading a Pre-defined Configuration Resource

Plugins are typically added to, and used by, a deployment template.

There may be cases when a custom plugin needs access, at runtime, to a specific resource such as a property file. One simple way to ensure the resource is always present and available for the plugin is to include the resource in the same template that the plugin is included in. Any resource can be added to a template using the ARE Web user interface. For information about how to add other resources to a deployment template, see the **Adding Other Resources to a Template** document on the ARE product Web site:

http://www.ibm.com/systems/power/software/i/are/documentation.html

Once resources have been added to the template, they are easily loadable and accessible to any plugin via the `getTemplateResource` method, which every plugin inherits from its BasePlugin base class.

For example, if we add myConfig.properties file into the template, by default, the template has the following structure:

Figure 16

The code of a plugin in this template can work like the following to retrieve the *myConfig.properties*:

```
protected ResultInfo runImpl() {

        …

        InputStream is = this.getTemplateResource("myConfig.properties");
        if (is != null) {
                Properties prop = new Properties();
                prop.load(is);
                ...
        }
        …

}
```

A more comprehend example for this feature is demonstrated by the *SimpleLoadResource* plugin, which is included in the sample plugins shipped with the ARE product.

Fixing Detected Problems

The ARE plugin architecture provides a mechanism for plugins to describe how to fix detected problems. This ability for ARE plugins to fix detected problems is commonly referred to as the 'AutoFix' framework.

Before describing this framework, it is very important to note that fixing problems using the ARE architecture is a two stage process:

1) Describe how to fix the problem
2) Using the information from stage 1, fix the problem

Stage 1 is what's done when a plugin is run during verification of a system using a template. Stage 2 is a *separate* step; it is something that occurs **after** verification. What

this means is that a custom plugin should **never** directly fix a problem. The purpose of a plugin is to do verification; if it detects problems that it wants fixed, the correct way to have those problems fixed is to use the AutoFix framework to describe how to fix the problem. The plugins provided by ARE already use the AutoFix framework to describe how to fix certain types of problems, such as file authority, object ownership, and certain user profile attributes. The process of actually fixing problems is performed *after* verification by invoking the areFix.sh script. This script utilizes a special "Auto Fix" plugin within the ARE Core that is capable of processing and fixing problems based on the descriptions about how to fix the problems. For more information about how to use the areFix.sh script to fix problem detected during verification, see the **How To Automatically Fix Problems** document on the ARE product Web site:

http://www.ibm.com/systems/power/software/i/are/documentation.html

As previously mentioned, a custom plugin has the capability to use the AutoFix framework to describe how to fix problems detected by the plugin. Describing how to fix a problem is done by creating a *fix action*. In the ARE environment, a fix action is represented by the `com.ibm.are.autofix.FixAction` class. To make fix actions easier to use, several sub-classes are provided by ARE; the most commonly used sub-class, which we will discuss shortly, is the `com.ibm.are.autofix.MethodFixAction` class.

The MethodFixAction class represents a description of how to fix a problem by invoking a Java method. Said another way, if you want to write a Java method to fix a problem detected by your plugin, the way to tell the AutoFix framework that your Java method should be used to fix the problem is by using the MethodFixAction class. A method fix action requires the following information:

- Problem category – A way to categorize problems. For example, if the file owner is incorrect, the problem category for its fix action may be something like "Owner" or "FileOwner".
- Problem description – A description of what the problem is.
- Name of fix method – This is the full class (including package) and method name of the **static** method that should be used to fix the problem.
- Fix method parameters – This is an array of Objects that represent each argument that needs to be passed to the fix method. If the fix method requires no parameters, you must pass a zero length array for this parameter.

There are a couple of items about the fix action information described above that warrant further discussion. First, note that the fix method must be a static Java method. The reason for this is that a static Java method is very easy to invoke; you do not need an object instance to invoke the method. This makes describing the fix method much simpler. If necessary, it would be possible to describe a fix method that is not static; this is possible using the `com.ibm.are.autofix.ReferenceFixAction` class. Usage of this class is not discussed in this document.

The other noteworthy item about the fix action information is the list of parameters to pass to the fix method. Since all fix actions are written into the XML report generated by ARE, the AutoFix framework must be able to describe, in pure XML, each parameter type and its value. This means the parameter types supported by ARE are limited; i.e. not just any Java object is supported. The AutoFix framework supports the following method parameter types:

- Strings (i.e. `java.lang.String` objects)
- boolean (primitive type)
- int (primitive type)
- long (primitive type)

In addition to these supported parameter types, the AutoFix framework also supports any Java class that implements the `com.ibm.are.xml.XmlSerializable` interface. By implementing this interface, you can enable any Java class to describe itself as XML and, conversely, materialize an instance of that class from an XML description of it.

Once you have created a description of how to fix a problem – i.e. created a FixAction (or subclass of it) object, you need to provide this information to ARE. This is done by passing the FixAction object as a parameter to one of the problem reporting methods (`reportError`, `reportWarning`, `reportInfo`). The following example shows how to use the MethodFixAction class to describe how to fix a problem by invoking a basic static Java method. The static method accepts one string parameter.

```
protected ResultInfo runImpl() {

        ...
        String targetFile = "/tmp/myApp.lck";

        if (new File(targetFile).exist()) {

                // Define a method fix action
                MethodFixAction fixAction = new MethodFixAction(
                        this.getLogger(),
                        //class & method name to fix the problem
                        this.getClass().getName() + ".fixMethodDeleteFile",
                        //Any parameters that are required by the fix method.
                        new Object[] { targetFile },
                        "File Lock", //descriptive problem category name
                        "A file lock that should have been cleaned up" //fix action description
                );

                // Report an error with our custom Fix Action just defined.
                reportError("File " + targetFile + " exists.", fixAction);
        } else {
                reportStep("File " + targetFile + " does not exist. Create the file manually and try this
demo again.");
        }
        ...
        return new ResultInfo(true);
}

/**
```

```
 * Fix method to delete a file.
 *
 * The method should return boolean to indicate whether the fix succeeds or not.
 * This return value is optional however; there is no requirement that
 * a fix method have a boolean return value.
 *
 * @return true if the fix action is successful, otherwise false.
 *
 */
public static boolean fixMethodDeleteFile (String name) {
        String msg = "Deleting file: " + name;
        FixActionMessageHelper.getPlugin().reportStep(msg);

        boolean success = new File(name).delete();

        return success;
}
```

The sample code above is a fairly simple example of defining a fix action and passing it along when reporting the problem. The fix method is also pretty simple; it takes a single parameter, which is the name of the file that should be deleted. One interesting aspect of the fix method is the usage of the FixActionMessageHelper class. This class exists for the sole purpose of enabling fix methods to easily report the status and result of the fix action performed. The FixActionMessageHelper class contains a static `getPlugin` method; this method returns a reference to the special "Auto Fix" plugin that is responsible for processing fix actions defined by plugins. This Auto Fix plugin is derived from the standard BasePlugin class, so once a reference to it is retrieved, all of the standard plugin features (logger, reporter, etc) are available for use. In the sample code above, we make use of the `reportStep` method to report the status of the fix method that is tasked with fixing the problem by deleting the specified file.

Using this simple example, you should now have a very good idea of how to create fix methods, and how to describe using that method to fix problems using the MethodFixAction class.

For more comprehensive examples, refer to the *SampleAutoFix* and *SampleAutoFix2* plugins, which are included in the sample plugins shipped with the ARE product.

Utility Classes and Methods

There are several classes in the ARE core that can be very useful when writing custom plugins. A brief description of the classes and their methods is provided below.

General Classes

General utilities are provided by classes in the arecore.jar. Since this jar is always included in the ARE runtime classpath, all of the utility classes and methods described below are available for use by custom plugins without any additional setup or runtime requirements. Note that we may not list all possible methods of interest for the classes; for a full list of available methods, as well as documentation for them, you can download javadoc for the ARE classes from the ARE product Web site:

http://www.ibm.com/systems/power/software/i/are/documentation.html

`com.ibm.are.common.CommandRunner` – Generalized, high level wrapper class for the `java.lang.Runtime.exec()` method.

`com.ibm.are.common.StringList` – This class extends the java.util.LinkedList class. Instead of allowing a heterogeneous list of items, this class forces all list entries to be strings. This allows the class to provide many useful methods for managing a list of strings, including case-insensitive versions of commonly used methods such as `contains`, `containsAll`, `equals`, `removeAll`, as well as methods to search the contents of the list for a particular substring.

`com.ibm.are.common.Utils` – Contains a variety of general purpose, static utility methods.

IBM i Related Classes

IBM i related utility classes are provided in the areibmi.jar. When developing a custom plugin, if you want to make use of IBM i utility classes, you will need to add areibmi.jar to your Java build path. However, no additional jars are needed when testing your plugin, or when using it as part of a template. This is because the areibmi.jar is already added to the ARE core runtime classpath by default.

With the correct HTTP group PTF level (see Compiling Your Plugin for required HTTP group PTF level) the areibmi jar can be found here:

/QIBM/ProdData/OS/OSGi/healthcheck/lib/areibmi.jar.

`com.ibm.are.platform.impl.IBMi` – Contains many different methods specific to IBM i, such as how to easily judge what OS release the plugin is running on, determine if a specific product or language is installed, determine if a certain PTF is applied, or retrieve a fully initialized, ready to use instance of the AS400 Toolbox class.

Here is an example below of how to retrieve a reference to the IBMi object and utilize its methods. In the example below, we make use of the IBMi object to quickly determine if we are running on an IBM i release that is prior to 6.1.

Example: Determining OS release

```
import com.ibm.are.platform.impl.IBMi;
…

        IBMi ibmi = IBMi.get();

        if (ibmi.isPriorToV6R1()) {
                //things to do when the OS is prior to 6.1
        }
```

Appendix

Sample

A sample is shipped with the ARE product. It is located at /QIBM/UserData/ARE/samples/sample.jar. The sample.jar by itself is a valid deployment template, with all source file packaged within the jar.