

How to Optimize and Encrypt Your XI Java Mappings

Applies to:

SAP NetWeaver Exchange Infrastructure (SP16)
SAP NetWeaver Application Server Java development (SP16)

Summary

This article describes the process of optimizing and encrypting SAP XI Java mappings. The process is based on a technique called "obfuscation." This consists of removing unused code and replacing Java classes, methods, and attributes with encrypted names. The code becomes harder to reverse-engineer, but the functionalities are unaffected by the changes.

Java obfuscation can be used for two main purposes:

1. To protect your intellectual property and hide critical business transformations from reverse-engineering.
2. To optimize and reduce the size of your code.

This article presents a step-by-step guide to obfuscate your SAP XI Java mapping and provides useful tools and resources.

Author(s): Lionel BIENNIER and Nicolas ADELIN

Company: TeamWork Management SA (Geneva - Switzerland)

Created on: 25 April 2006

Authors Bio



Lionel is a SAP XI Architect / Consultant. He is in charge of SAP XI and New Technologies' team at TeamWork Management SA. His focus is to design and deliver innovative and efficient business process integration solutions. He is trained in Solution Manager. He is certified in SAP Exchange Infrastructure, SAP Sales and Distribution and SAP ABAP.



Nicolas is a SAP consultant at TeamWork Management SA. He is trained and experienced in SAP XI development and configuration. He concentrates on advanced SAP XI Java developments. He also has experience with WebDynPro and J2EE application development.

Table of Contents

How to Optimize and Encrypt Your XI Java Mappings.....	1
Applies to:	1
Summary.....	1
Authors Bio	2
Table of Contents	2
Why Obfuscate Your Java Code?	3
Prerequisites.....	3
Overview of the Obfuscation Process	3
Installing the Sample XI Java Mapping	4
Obscuring the SAP XI Java Mapping	9
Comparing Java Code Before and After Obfuscation	11
Conclusion	13
Disclaimer and Liability Notice.....	14

Why Obfuscate Your Java Code?

You might want to obfuscate your SAP XI Java programs in order to:

- Hide critical business logic and transformations,
- Protect your source code against reverse-engineering,
- Prevent unauthorized patches and modifications,
- Optimize and shrink the size of your Java code,
- Increase security by encrypting log on, authorization code sections.

Prerequisites

The following tools are required:

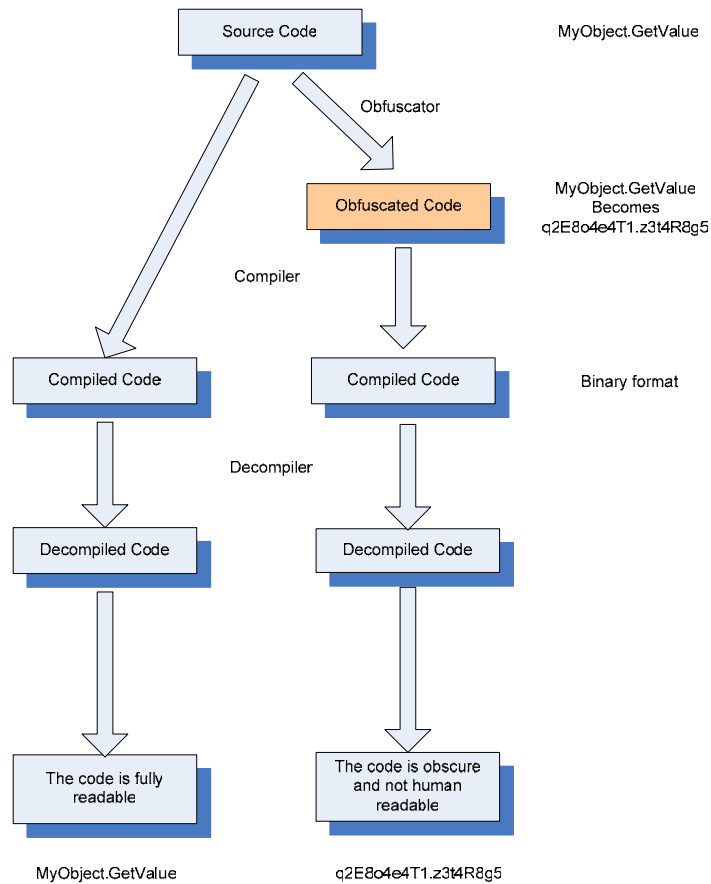
- A Java obfuscator tool. A list of available tools is given in the section “Obscuring the SAP XI Java mapping.”
- SAP NetWeaver Developer Studio

Overview of the Obfuscation Process

The obfuscation process consists in translating explicit names and syntax into meaningless names.

A basic example will illustrate this: if your source code contains a method named “getValue” it can be encrypted into “e1zT3Dw%2” or “z” or “3ert” or any other alphanumeric string. The logic and functionalities of the method “getValue” are unaltered but the name of the method is no longer human-readable.

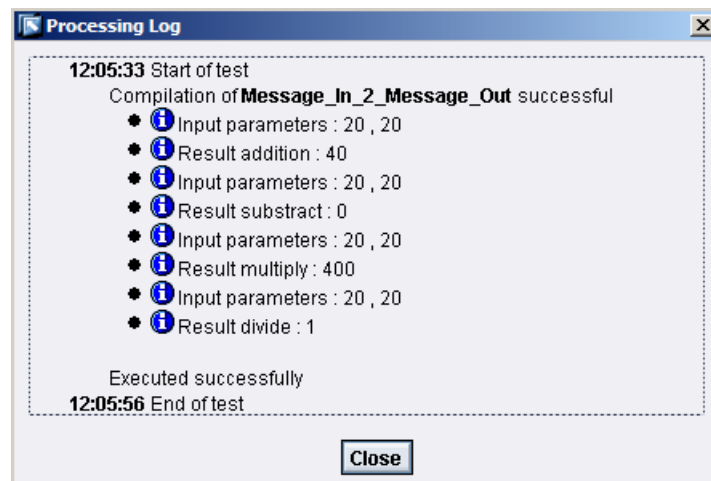
The diagram below shows the process of reverse-engineering a Java compiled code. On the left track without obfuscation the reverse-engineering is successful. On the right track the code cannot be reverse-engineered.



Installing the Sample XI Java Mapping

We developed a sample Java mapping program to showcase the obfuscation technique in SAP XI.

The Java mapping program initializes two values (initFirstValue and initSecondValue) and performs four calculations (addTwoValues, divideTwoValues, multiplyTwoValues, subtractTwoValues). The result of each operation is displayed in the SAP XI processing log window.



To get started with the sample program you need to start the SAP Netweaver Developer Studio, then create the packageExample and copy the source code of SampleCalculation and OperatorClass in two separate classes.

```
package packageExample;

import com.sap.aii.mapping.api.MappingTrace;
import com.sap.aii.mappingtool.tf3.rt.Container;

public class SampleCalculation {
    public SampleCalculation () {
    }
    public void calculationLogic(Container c) {
        MappingTrace trace;
        trace = c.getTrace();

        int result;

        try {
            //Create an instance of OperatorClass
            OperatorClass myOperator = new OperatorClass();
            //Init the values for the operation
            int firstValue = myOperator.initFirstValue();
            int secondValue = myOperator.initFirstValue();

            //Log the input parameters into the trace

            trace.addInfo(myOperator.writeInputParameters(firstValue,secondValue));
            //Addition
            result = myOperator.addTwoValue(firstValue,secondValue);
            //Log the result into the trace
            trace.addInfo(myOperator.writeResult(OperatorClass.addition,
result));

            //Log the input parameters into the trace

            trace.addInfo(myOperator.writeInputParameters(firstValue,secondValue));
            //Substraction
            result = myOperator.subtractTwoValue(firstValue,secondValue);
            //Log the result into the trace
            trace.addInfo(myOperator.writeResult(OperatorClass.subtract,
result));
```

```

        //Log the input parameters into the trace

trace.addToInfo(myOperator.writeInputParameters(firstValue,secondValue));
        //Multiply
        result = myOperator.multiplyTwoValue(firstValue,secondValue);
        //Log the result into the trace
        trace.addToInfo(myOperator.writeResult(OperatorClass.multiply,
result));

        //Log the input parameters into the trace

trace.addToInfo(myOperator.writeInputParameters(firstValue,secondValue));
        //Division
        result = myOperator.divideTwoValue(firstValue,secondValue);
        //Log the result into the trace
        trace.addToInfo(myOperator.writeResult(OperatorClass.divide,
result));

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

package packageExample;

public class OperatorClass {

    static String addition = "addition";
    static String subtract = "subtract";
    static String multiply = "multiply";
    static String divide = "divide";

    public OperatorClass () {

    }

    //Init the first value

```

```

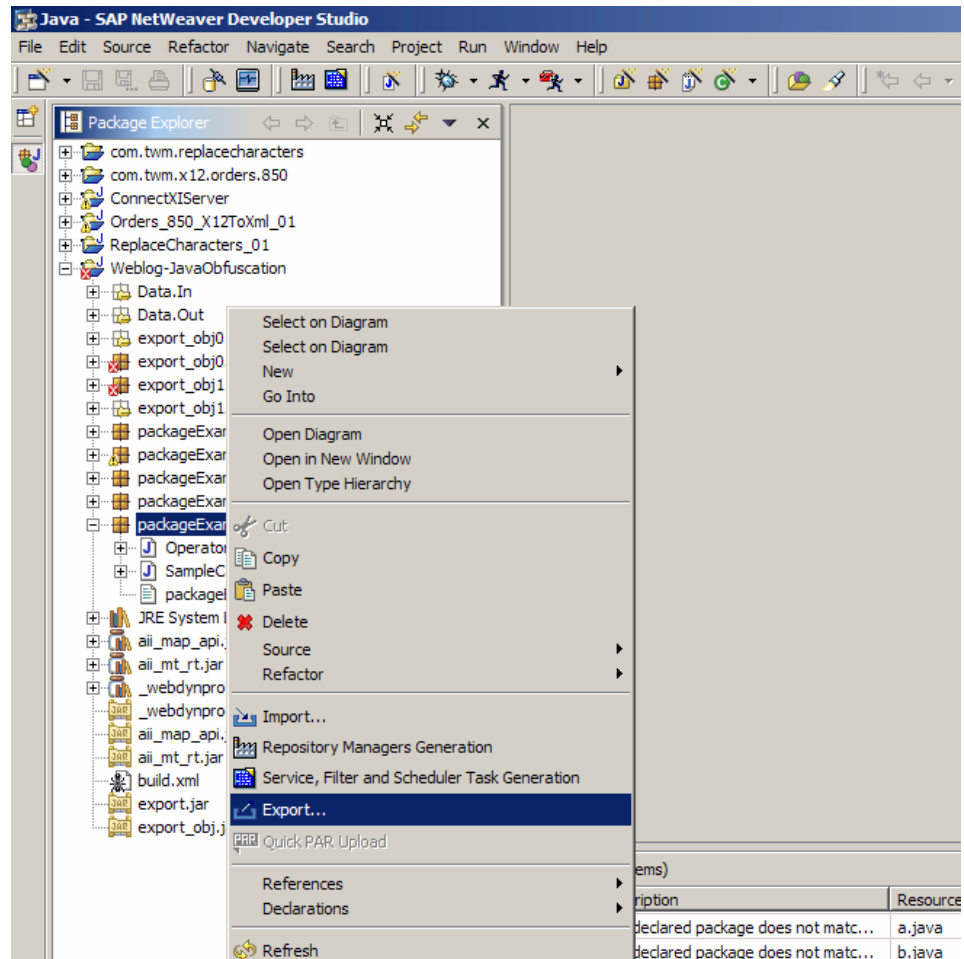
public int initFirstValue() {
    return 20;
}
//Init the Second value
public int initSecondValue() {
    return 15;
}
//Addition between two values
public int addTwoValue (int value1, int value2) {
    int result=0;
    result = value1 + value2;
    return result;
}
//Subtraction between two values
public int subtractTwoValue (int value1, int value2) {
    int result=0;
    result = value1 - value2;
    return result;
}
//Multiplication between two values
public int multiplyTwoValue (int value1, int value2) {
    int result=0;
    result = value1 * value2;
    return result;
}
//Division between two values
public int divideTwoValue(int value1, int value2) {
    int result=0;
    result = value1 / value2;
    return result;
}
//Returns the string result for the logging
public String writeResult(String operator, int result) {
    return "Result " + operator + " : " + result;
}
//Returns the string input parameters for the logging
public String writeInputParameters(int firstValue, int secondValue) {
    return "Input parameters : " + firstValue + " , " + secondValue ;
}

```

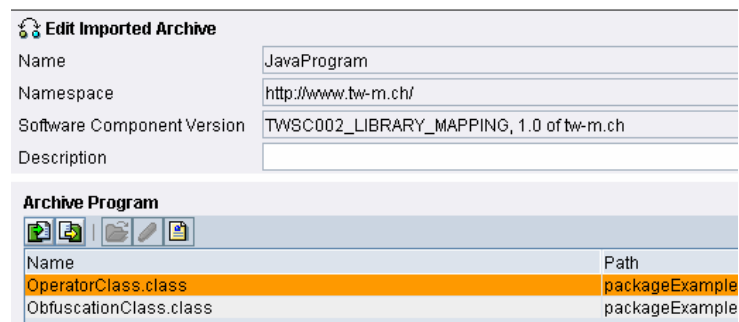
}

The procedure below describes the steps to export the Java mapping (packageExample) from SAP Netweaver Developer Studio and import it into SAP XI.

Export the Java source code from SAP Netweaver Developer Studio as a jar file.



Create SAP XI archive and import the jar file.



<p>Create a message mapping.</p>	 <p>The screenshot shows the SAP XI Message Mapping Designer interface. At the top, there are two message trees. The left tree is for 'Message_In' and the right is for 'Message_Out'. Both trees show a 'DATA' element with an occurrence of '1..1' and a type of 'xsd:string'. Below the trees, a mapping diagram shows a 'UserDefine...' function block connected to a 'ROW' block. At the bottom, the 'Functions' dropdown menu is set to 'User-Defined'.</p>
<p>Create a user defined function.</p>	<p>Description:</p> <p>Imports packageExample2_save.*;</p> <pre> public String UserDefinedFunction(Container container){ SampleCalculation mySampleCalculation = new SampleCalculation(); mySampleCalculation.calculationLogic(container); return "test"; } </pre>
<p>Go to the test tab and execute the mapping.</p>	 <p>The screenshot shows the 'Processing Log' window. It displays the following test results:</p> <pre> 12:05:33 Start of test Compilation of Message_In_2_Message_Out successful * Input parameters : 20 , 20 * Result addition : 40 * Input parameters : 20 , 20 * Result subtract : 0 * Input parameters : 20 , 20 * Result multiply : 400 * Input parameters : 20 , 20 * Result divide : 1 Executed successfully 12:05:56 End of test </pre> <p>A 'Close' button is visible at the bottom right of the window.</p>

Obscuring the SAP XI Java Mapping

Numerous commercial and freeware obfuscator tools are available on the market, including:

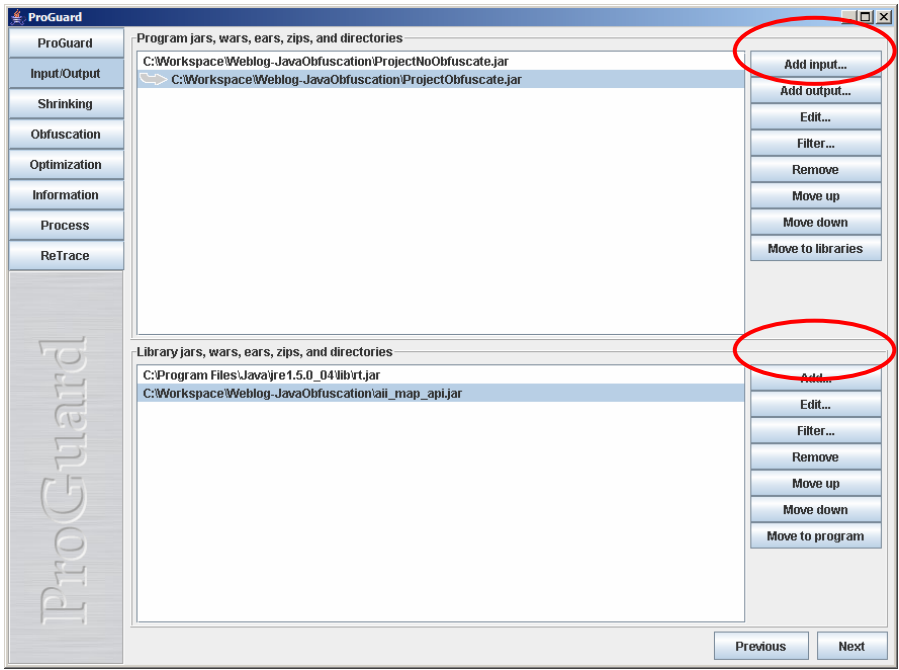
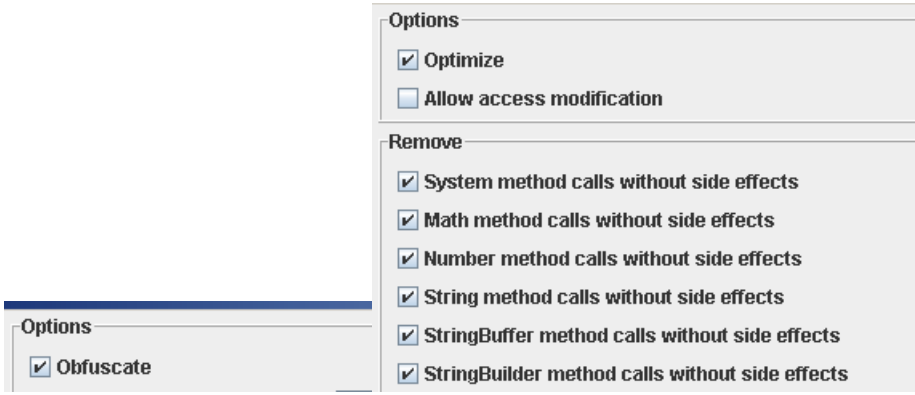
Proguard

Retrologic

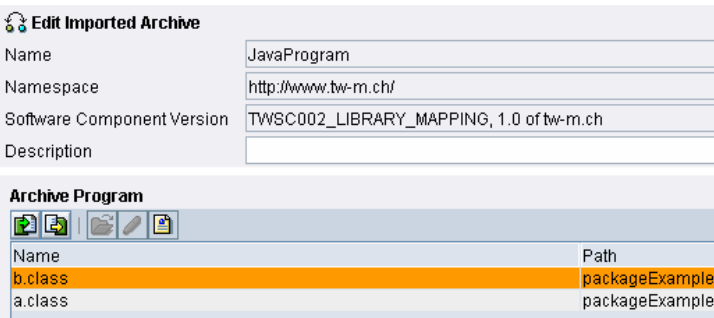
[IBM AlphaWorks](#) (JAX),
[PreEmptive](#) (DashOPro),
[Zelix](#) (KlassMaster),
[S5 Systems](#) (iPresto).

We will use a freeware called ProGuard. The tool is easy to use and it has a graphical user interface.

The procedure below describes the two steps required to obfuscate the Java mapping file.

<p>Add the Java mapping jar as input source and give the name of the output.</p> <p>Add the SAP XI logging aii_mt_rt.jar for the Java library jars.</p>	
<p>Check the options Obfuscate, Optimize and press Execute.</p>	

The Java mapping file is obfuscated and it can be imported into SAP XI:

Open the previously created SAP XI archive and import the obfuscated jar file.	
Modify the user defined function to match the obfuscated class name and method.	

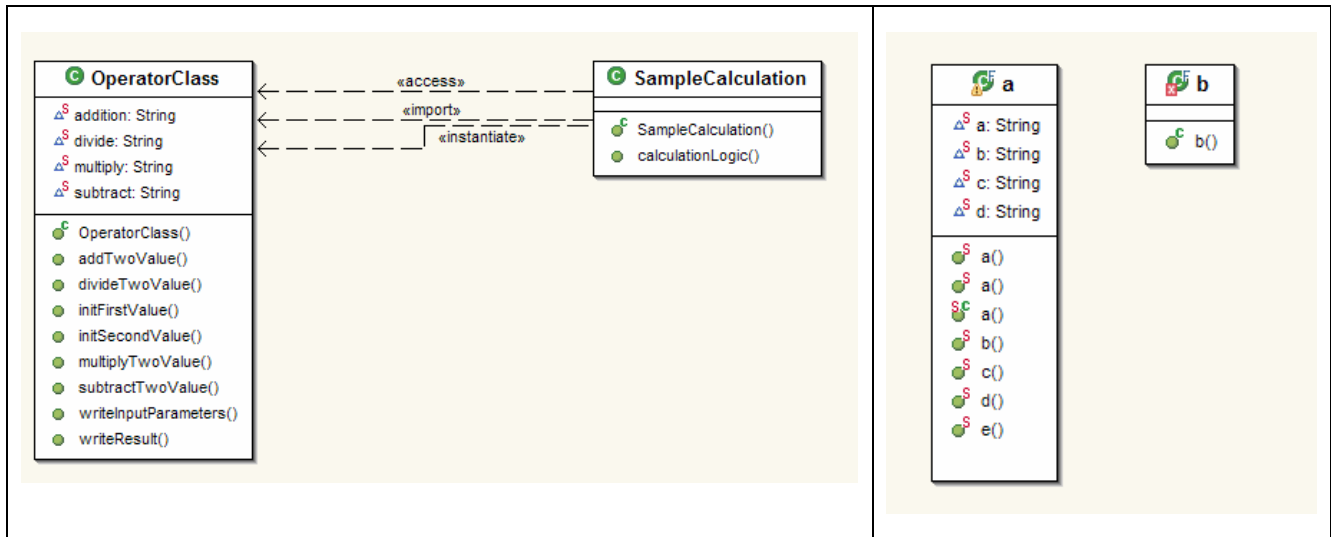
When you go to the test tab and run the mapping again the behavior is unchanged and the processing log window shows the same results.

Comparing Java Code Before and After Obfuscation

Class Diagrams

The UML class diagram of the source code before and after obfuscation shows the transformation operated on classes, methods and attributes names. Names were replaced by a single letter.

UML Class Diagram before obfuscation	UML Class Diagram after obfuscation
---	--



Before and After Comparison

The table below highlights the main changes in the source. The class, methods and attributes names are replaced by a letter. The developer comments are removed (see line 7). Unused class, methods or attributes are also removed.

	Before obfuscation	After obfuscation
1	<code>public class SampleCalculation {</code>	<code>public final class b {</code>
2	<code>public void calculationLogic(Container c)</code>	<code>private static void a(Container container)</code>
3	<code>int firstValue = myOperator.initFirstValue();</code>	<code>int j = packageExample3.a.a();</code>
4	<code>result = myOperator.addTwoValue(firstValue,secondValue);</code>	<code>int l = packageExample3.a.a(j, k);</code>
5	<code>trace.addInfo(myOperator.writeResult(OperatorClass .addition, result));</code>	<code>abstracttrace.addInfo(packageExample3.a.a(a, a, i));</code>
6	<code>static String addition = "addition";</code>	<code>static String a = "addition";</code>
7	<code>//Addition between two values</code> <code>public int addTwoValue (int value1, int value2) {</code>	<code>public static int a(int i, int j) {</code>

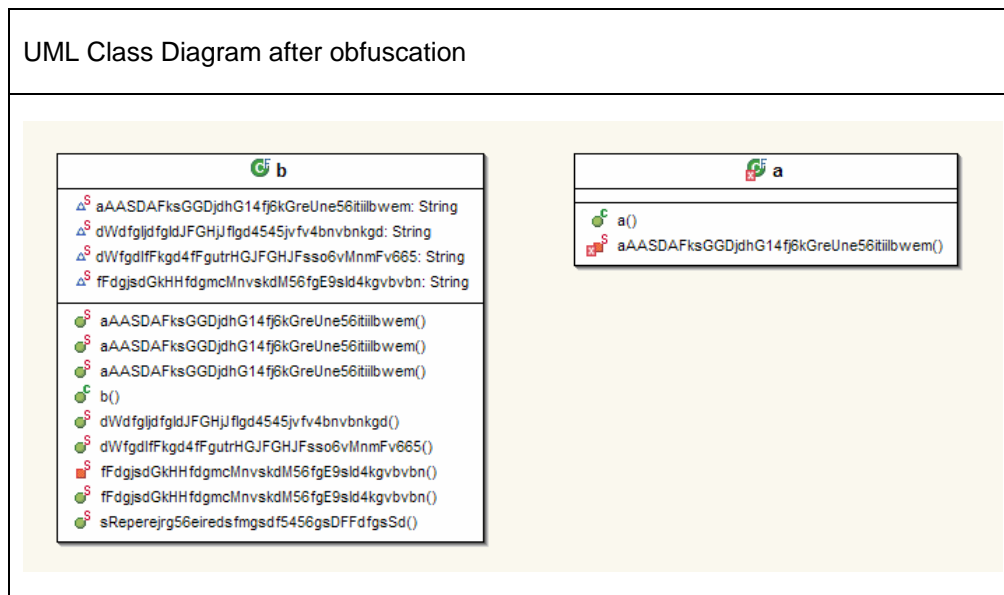
Optimization

The obfuscation process will optimize the Java code by reducing the source code size. The reduction percentage depends on the Java code itself: number of classes, instances and methods, length of classes, instances and methods, number and length of comments, number of unused methods. To give some empirical figures, the Java code of a complex Java mapping program can be reduced by 30 to 40%.

We also observed a positive effect on run time performances. The execution time of optimized/obfuscated Java transformations was faster by 10 to 20%.

Advanced Obfuscation

You can tailor the behavior of the obfuscation tool to match your specific needs. For example, we increased the level of encryption by removing the optimization option and customizing the obfuscation dictionary. The result is shown below.



Conclusion

This article presented the obfuscation process and highlighted the main features such as encryption and optimization. The insertion of obfuscation into the existing SAP XI development processes is another challenge. It requires appropriate thinking, preparation and management. Another learning of this article is that proven technologies in the Java environment can be easily transferred and reused in SAP XI Exchange Infrastructure.

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.