

How to use the JSourceObjectizer - some short instructions

Wednesday, 02 April 2008

Last Updated Friday, 21 September 2012

This document provides some information about how the JSourceObjectizer can be used by the following sections:

-

What's needed to use the JSourceObjectizer

-

How to parse a Java source and get it's JSOM tree

-

Some information about the JSOM types

-

Two possibilities of walking through a JSOM tree

-

Comparing Java source fragments (or even complete source files)

-

Some tips and tricks

What's needed to use the JSourceObjectizer

The JSourceObjectizer distribution contains a directory called lib with three jar files:

- the JSourceObjectizer itself
- an ANTLv3 runtime distribution

- a HSD core library

To use the JSourceObjectizer please ensure that all these three jar files have been included into your project's build path, however this may look like in your project/build environment. This is just enough to be prepared for using the

JSourceObjectizer.

Furthermore, the JSourceObjectizer needs Java 1.5 or later.

[Back to the top](#)

How to parse a Java source and get it's JSOM tree

One of the shortest way to parse a Java source code with the JSourceObjectizer could look like this:

```
import com.habelitz.jsoobjectizer.unmarshaller.JSourceUnmarshaller;
import com.habelitz.jsoobjectizer.jsom.api.JavaSource;

// args[0] must state the path and file name of a Java source file.
public static void main(String[] args) {
    JavaSource javaSource = null;
    try {
        javaSource = new JSourceUnmarshaller().unmarshal(new File(args[0]), null);
    } catch (JSourceUnmarshallerException e) {
        e.printStackTrace();
    }
    if (javaSource != null) {
        // do something with it
    }
}
```

The method `JSourceUnmarshaller.unmarshal()` returns an object of type `JavaSource`, which is the JSOM type that represents the root of a parsed Java source. The next section deals with some fundamentals about JSOM types.

[Back to the top](#)

Some information about the JSOM types

The base type JSOM

The type JSOM (Java Source Object Model) is the supertype of all types that represent a more or less complex part of a Java source. For instance, the type `JavaSource` represents a complete Java source file with all its type declarations and so on. On the other hand the type `ModifierList` represents just a list of modifiers that may be stated with method declarations or local variable declarations. But both, `JavaSource` and `ModifierList` are subtypes of the type JSOM.

Normally a supertype shouldn't know of what subtype it is. The JSOM base type breaks this rule in order to avoid a large amount of instanceof operations. Therefore, the interface JSOM defines enum constants for a lot of JSOM subtypes and the method `isJSOMtype(JSOM.JSOMtype pType)` that can be used to find out if a JSOM object is of a certain JSOM subtype. But don't panic: there're a lot of JSOM getter methods that already return an object of a concrete JSOM type. For instance, the JSOM type `ClassTopLevelScope` does not just declare a (fictive) method like `getContent()` which returns a load of plain JSOM objects and that lets it up to the caller to find out of what JSOM types the returned objects are. Instead, `ClassTopLevelScope` defines methods like `getMethodDefinitions()` or `getConstructorDefinitions()` that return lists of objects which are already of type `MethodDefinition` and `ConstructorDefinition` respectively. BTW: The JSourceObjectizer V2.xx, which is currently under development, will provide much more of such getter methods that return a certain subset of all the members a JSOM type may have.

However, there's still one important point regarding the constants defined by the enum JSOM.JSOMType that should be explained here. JSOM.JSOMType defines only one constant for all kinds of expressions and one for all kinds of statement block elements(1): JSOMType.EXPRESSION and JSOMType.STATEMENT_BLOCK_ELEMENT. Therefore, the call `anyJSOMObject.isJSOMType(JSOM.JSOMType.EXPRESSION)` only tells if a JSOM object represents an expression but not of what kind of expression it is (if it is an expression at all). The types `Expression` (the base type of all JSOM types representing an expression) and `StatementBlockElement` (the base type of all JSOM types representing a statement block element) define their own enum constants for their various subtypes. But, nevertheless, the JSOM interface defines the methods `isExpressionType(Expression.ExpressionType pType)` and `isStatementBlockElementType(StatementBlockElement.ElementType pType)`. If it should be checked whether a JSOM object is of a certain expression type or statement block element type it's not necessary to call the method `isJSOMType(...)` first to find out if the JSOM object represents an expression or statement block element at all.

1) A statement block element is something that can be stated within a statement block scope, i.e. within a method scope or a compound statement scope, for instance; therefore, a statement block element can be a statement, a local variable declaration or a local type declaration.

The JSOM interface types

Once a Java source has been parsed a JSOM object of type `JavaSource` will be returned by the `JSourceObjectizer`'s `unmarshaller` (see the code example above). This `JavaSource` object is the entry point for walking through the complete parsed Java source. To get a feeling for the JSOM interface types it's recommended to skim through the `JSourceObjectizer`'s API documentation which is part of the downloadable `JSourceObjectizer` distribution. And it's also recommended to ignore all the content that can be found for the package `com.habelitz.jsobjectizer.unmarshaller` and its subpackages.

The following list describes the content of the packages that are most relevant to `JSourceObjectizer` users:`com.habelitz.jsobjectizer.jsom`

This package contains the interface `JSOM`, i.e. the base type of all JSOM types. This interface contains an inner enum declaration `JSOMType` which defines constants for all JSOM types excepting constants for all concrete expression and statement block element types.

-

`com.habelitz.jsobjectizer.jsom.api`

All JSOM types representing a certain element that can exist within a Java source and that have nothing to do with expressions or statement block elements can be found within this package.

`com.habelitz.jsobjectizer.jsom.api.base`

Some JSOM types have a common basis and differ only in some aspects. For instance, both a class' top level scope and an interface's top level scope can have abstract method declarations, field declarations and so on. But an interface top level scope never has a constructor definition. Therefore the JSOM types `ClassTopLevelScope` and `InterfaceTopLevelScope` are not derived from the type `JSOM` directly. Instead, the common features of such JSOM types are bundled together by appropriate base types. Such base types can be found within this package.

`com.habelitz.jsobjectizer.jsom.api.expression`

This package contains all the JSOM types that deal with expressions. The base type for all kinds of expressions is `Expression`. This base type also contains an inner enum declaration `ExpressionType` which defines constants for all concrete expressions.

-

`com.habelitz.jsobjectizer.jsom.api.statement`

This package contains all the JSOM types regarding statement block elements, i.e. statement types and types for local variable declarations and local type declarations. The base type for all kinds of statement block elements is

StatementBlockElement. This base type also contains an inner enum declaration ElementType which defines constants for all concrete statement block elements.

The implementation of the JSOM interface

All the JSOM interface types are implemented by the content that can be found within the package `com.habelitz.jsojectizer.unmarshaller.jsombridge` and its subpackages. For every JSOM interface type there's also an appropriate implementing (maybe abstract) class within one of these packages, with the same name as the JSOM interface type but prefixed by AST2. For example, the interface `com.habelitz.jsojectizer.jsom.api.statement.ReturnStatement` is implemented by the class `com.habelitz.jsojectizer.unmarshaller.jsombridge.ast2api.statement.AST2ReturnStatement`.

Unless you need to change or improve these implementations it's recommended simply to ignore all the AST2Whatever stuff.

[Back to the top](#)

Two possibilities of walking through a JSOM tree

There are two possibilities of walking through a JSOM tree:

-

the manual way

-

using the JSOMTraverser call back methods

Walking through a JSOM tree manually

Nearly all JSOM type interfaces provide getter methods to get the content or elements of a certain JSOM type and it's possible to walk through a JSOM tree only by using these methods. But they should be used very carefully because a lot of them may also return null. This is the case for all the content or elements that are optional for a certain JSOM type and that doesn't actually exist. For instance, if `anyMethodDecl` is an object of the JSOM type `MethodDeclaration` representing a method declaration that has no formal parameters and if you call `anyMethodDecl.getFormalParameters()` you will not receive an empty object of type `FormalParameterList` but simply null instead.

On the other hand, there are also a lot of such getter methods that never return null because they return an obligatory content or element of a certain JSOM type. Therefore it's not a bad idea to have a look into the `JSourceObjectizer`'s API documentation if you are unsure whether a getter method returns an obligatory or optional content.

Walking through a JSOM tree by using the JSOMTraverser

About the JSOMTraverser

For most cases using the JSOMTraverser should be the preferred way for traversing a JSOM tree. This can reduce the problems regarding getter methods that may return null (see the paragraphs above) enormously. Furthermore, traversing a JSOM tree manually can become a fiasco.

The base type of the JSOMTraverser is the interface `TraverseAction`,

which can be found within the package `com.habelitz.jsojectizer.jsom.util`. The interface `TraverseAction` declares the following pair of methods for most JSOM types:

-
`performAction(AnyJSOMType pJSOMType)`

-
`actionPerformed(AnyJSOMType pJSOMType)`

While traversing a Java source by the `JSOMTraverser` the method `performAction(...)` will be called immediately when a certain JSOM type object becomes the current traversing candidate and with this object as argument. The method `actionPerformed(...)` will be called just after a certain JSOM type object and all its content and elements, that are of type JSOM, too, have been traversed.

What should happen within these methods is up to the individual needs, of course. Because it's very likely that not all of the `performAction(...)` and `actionPerformed(...)` methods are of interest for a certain task there's also an adapter class `TraverseActionAdapter` that implements all the `TraverseAction` methods but with empty method bodies.

How to start traversing with a `JSOMTraverser`

The interface `JSOM` declares the method `traverseAll(TraverseAction pAction)` and is therefore available for all JSOM types. If this method gets called on any JSOM type object the following will happen in the stated order:

-
At first the method `pAction.performAction(this)` will be called.

-
Then, if the current JSOM type object has any JSOM type members, the method `traverseAll()` will be called on all of these members with the `TraverseAction` object passed to the initial `traverseAll()` call as argument.

-
Finally the method `pAction.actionPerformed(this)` will be called.

Traversing with a `JSOMTraverser` can be started with any JSOM type object. If it gets started with an object of the JSOM type `JavaSource` all the content of a Java source will be traversed, of course.

[Back to the top](#)

Comparing Java source fragments (or even complete source files)

Overview

Most JSOM types (the most concrete ones or their super types or even both) declare a `compareTo()` method to compare JSOM objects of the same JSOM type. This comparison isn't done just lexically. For instance, such a comparison also takes into account if the order of compared JSOM sub-elements, that belong to certain compared JSOM root objects, is arbitrary or not.

As a rule of thumb, if the order of two lists of JSOM sub-elements is arbitrary (modifier list, method declarations, import statements and so on), the comparison checks if each element from one list has an equivalent within the other one and reports missing elements. On the other hand, if the order isn't arbitrary (statement list, formal parameter list, ...), the sub-elements at the same position will be compared.

There're three main types used to report found differences:

- `SemanticDelta` (an interface)

Objects of that type represent comparison differences and are returned by the `compareTo()` methods

- `SemanticDeltaDescription` (an interface)

Objects of this type provide a textual description for a certain comparison difference; the message text can be customized.

- `Delta` (enum types that implement the `SemanticDeltaDescription` interface)

For each possible (or at least supported) difference there's an appropriate enum constant.

`Delta` and `SemanticDeltaDescription`

A lot of JSOM interface types declare an inner `Delta` enum type defining constants for various kinds of differences that may be found when comparing two JSOM objects of the same type. If a certain JSOM type doesn't contain such an inner `Delta` enum the reason for that and most likely some hints and tips are described within the type's main comment. For the most time (if not always) a user of the `compareTo()` feature hasn't to deal with the `Delta` enum constants directly. However, there're two exceptions:

At first these enum declarations implement the interface `SemanticDeltaDescription` and the default description messages are just the identifiers of the enum constants. But, as said within the list above, the message texts are customizable - simply via calling the method `setDescription(String)` on a certain `Delta` enum constant.

Secondly a description might be a little bit cryptic (at least if not customized - but maybe even then :-)) . If the reason for a found difference isn't quite clear it may help to view the appropriate `Delta` enum constant declaration - they're well documented what hopefully helps in such situations.

`SemanticDelta`

A list of objects of this type is returned by the `compareTo()` methods if differences have been found (but null otherwise). A `SemanticDelta` object provides the following information:

- The compared root JSOM objects that have been recognized as being different.
- The descriptions of found differences.
- The different JSOM sub-elements of the compared JSOM root objects.

The actual difference or differences of compared JSOM objects may be deeply nested. For example, if two complete and large Java source files have been compared and if there're differences within deeply nested localscopes, the root JSOM objects returned from the root `SemanticDelta` object's methods `getComparedJSOM()` and `getComparedToJSOM()` are the compared `JavaSource` objects. Then the `SemanticDelta` methods returning the sub-deltas provide information regarding the method/constructor declarations or static/instance initializers that contain the differences. These sub-deltas then provide further sub-deltas themselves for the different method/constructor (or whatever) scopes and so on and so forth up to the nested scope that finally contains the statements and expressions that are different, i.e. this continues until the last bunch of sub-deltas has been reached that deal with the actual differences.

This means, the comparison actually builds `SemanticDelta` trees with maybe tight and/or long branches. Because it's a thankless task to walk such a `SemanticDelta` cascade manually the `SemanticDelta` type provides the method `getLeaves()` which, when called on returned root `SemanticDelta` objects, collects all the informations of the actual differences and filters all the noise from and between a root `SemanticDelta` up to the leaves representing the actual differences.

[Back to the top](#)

Some tips and tricks

Partially disabling and (re)enabling the traversal of JSOM type members

The JSOMTraverser interface TraverseAction also defines the methods isMemberTraversingEnabled() and setMemberTraversingEnabledState(boolean pNewState). These methods are already implemented by the TraverseAction adapter class TraverseActionAdapter and the default value is true.

While traversing a JSOM tree with a JSOMTraverser the members of a JSOM type object will only be traversed if isMemberTraversingEnabled() == true. This means that traversing such members can be disabled and reenabled at any time. For instance, if the content of all statement block scopes are out of interest it would be a waste of time to traverse them. To disable traversing the stuff inside a statement block scope simply override the methods performAction(StatementBlockScope) and actionPerformed(StatementBlockScope) within your TraverseAction implementation as follows:

```
public class MyJSOMTraverser extends TraverseActionAdapter {

    // ... my individual content ...

    @Override
    public void performAction(StatementBlockScope pStatementBockScope) {
        setMemberTraversingEnabledState(false);
    }

    @Override
    public void actionPerformed(StatementBlockScope pStatementBlockScope) {
        setMemberTraversingEnabledState(true);
    }

    // ... my individual content ...

}
```

Disabling the traversal saves not only time but also memory because JSOM types are generated on demand by the JSourceObjectizer. As a matter of fact, after parsing a Java source only the JSOM type JavaSource will be created. All other JSOM types will only be created when an appropriate getter method gets invoked and if there's something that should get returned.

Nested usage of different JSOMTraversers

The nested usage of different TraverseAction implementations can simplify the one or another lookahead job. If one of your performAction() methods has been invoked it could be the case that you need some information about the members of the currently traversed JSOM type object. But no information about these members can exist at this point of time. Just waiting until the matching actionPerformed() method gets called should be the best choice for most cases but in some situations waiting until the members have been traversed could rise some problems.

To give an example for such a problem, imagine that you have to do some tasks within the TraverseAction method performAction(MethodCall pMethodCall). Furthermore imagine that in order to work on these tasks you need some information about the arguments stated with the method call pMethodCall. Traversing the arguments manually could possibly be the right choice. But an argument can also be a very complex expression and traversing such expression manually could be the hell. At the first view it seems to be simple just to let the traverser walk through the complete method call, collect the needed information about the method call's arguments and do the tasks regarding the method call within the method actionPerformed(MethodCall pMethodCall). This will work for method calls that have no method call as argument. But as soon as there're arguments in form of method calls all the collected information about the arguments must be stacked in some way.

To stack all the information about arguments will solve the problem but there's an alternative way. Within the method performAction(MethodCall pMethodCall) an instance of another TraverseAction implementation could be created that does nothing else then the lookahead job and that collects all the needed information just in time. Now call pMethodCall.traverseAll() with a temporary TraverseAction object, that does the lookahead job, as argument. That's it.

[Back to the top](#)