cssbox.sourceforge.net

Manual - jStyleParser - Java CSS parser

Radek Burget, burgetr@fit.vutbr.cz

15-19 minutes

Introduction

jStyleParser is a Java library for parsing CSS style sheets and assigning styles to the HTML or XML document elements according to the W3C CSS 2.1 specification and a subset of the CSS 3 specification. It allows parsing the individual CSS files as well as computing the efficient style of the DOM elements.

Style Sheet Parsing

This functionality may be used for parsing individual style sheets obtained from a remote file (URL), local file or a string. Three static methods are defined in the `CSSFactory` class for this purpose:

- `StyleSheet parse(URL url, String encoding)`, the most general method. Transforms data available at the given `url`, expecting given `encoding`, or taking the default one if it is not provided.,

- `StyleSheet parse(URL url, NetworkProcessor network, String encoding)` that has the same effect as above with providing a custom `NetworkProcessor` (see below),
- `StyleSheet parse(String fileName, String encoding)`, which internally transforms the `fileName` into an URL and
- `StyleSheet parseString(String css, URL base)`, which can be used to parse embedded CSS declarations that is declarations between the <style> tags.

During the parsing process, the parser automatically imports all the style sheets referenced using the `@import` rules. See the Media section for further reference about how to limit this behavior to certain media only or disable it completely.

For obtaining the imported style sheets referenced by their URLs, it is possible to provide a custom implementation of a `NetworkProcessor` that simply provides obtaining an `InputStream` from the given URL. The default built-in `NetworkProcessor` is based on the standard Java `URLConnection` infrastructure.

The result of the parsing is a `StyleSheet` object that is generally a collection of *rules* discovered in the style sheet. It can be also used as the input for the DOM `Analyzer` as described below in the DOM Analysis section.

Parsed style sheet processing

When the DOM Analyzer feature is not used, the parsed style sheed may be manually traversed in order to obtain the contained rules and declarations. The most important data structures are the following:

- The `StyleSheet` is a collection of *rules*.
- Each rule is an instance of `RuleBlock`. There exist several subclases of this class that correspond to the individual CSS rule types. The most important of them are `RuleSet` (a standard CSS rule with a selector and declarations) and `RuleMedia` (the `@media` rule containing further `RuleSet` rules).
- The `RuleSet` is a collection of `Declaration` objects. The selectors used for the whole rule may be obtained using the `getSelectors()` method.
- Each `Declaration` is a set of `Term` objects that represent the values that are assigned to the given property. There exist many subclasses of `Term` that represent the individual value types in CSS (lengths, integers, colors, identifiers, etc.) The name of the property can be obtained using its `getProperty()` method.

A simple example of the style sheet processing is available in the test.ParserDemo example.

Obtaining the style sheets used in an HTML document

The `CSSFactory` provides a method that analyzes an HTML or XML document represented by a DOM and it parses all the referenced style sheets:

- `StyleSheet CSSFactory.getUsedStyles(Document doc, String encoding, URL base, MediaSpec media)`

It parses all the style sheets referenced from the document that correspond to the specified media and it returns a single style sheet structure that contains all the relevant CSS rules. The `base` URL is used for the possible relative URLs used in the style sheets. The `encoding` specifies a default encoding that is used when the encoding is not specified within the style sheet. Finally, `media` specifies the current media features that should be used for evaluating the media queries used in the style sheet as described in the Media section.

DOM Analysis

The purpose of the DOM analysis is to apply the relevant style sheets to a particular HTML or XML document represented by a DOM in order to obtain the efficient styles of the individual document elements. The result of the analysis is a mapping between the DOM elements and the corresponding CSS declarations. This mapping can be used either for displaying the HTML document or for performing some further analysis on the document structure.

Analyzing a Style Sheet

When a `StyleSheet` instance is obtained from the parser, it can be passed to an `Analyzer`.

The `Analyzer` basically provides the following method:

- `StyleMap evaluateDOM(Document doc, MediaSpec media,                              boolean inherit)`, which constructs a map between DOM elements and their CSS declarations according to the given `media` specification and allowed/disabled inheritance of declarations.

Additionally, a `DirectAnalyzer` analyzer class is provided for cases when it is not necessary to evaluate the whole DOM. It computes a style for individual DOM nodes without creating the whole map. It is suitable for obtaining the style of individual elements without computing the style for the whole DOM tree. However, in larger scale, the performance of the individual computation is significantly worse.

Simplified and Direct Usage Method

To provide simpler approach while parsing an (X)HTML document, `CSSFactory` provides the following method:

- `StyleMap assignDOM(Document doc, String encoding,
                                  URL base, MediaSpec media, boolean useInheritance)`

It automatically downloads and parses all the internal and external style sheets referenced from a DOM for the given media and runs the style mapping to the given DOM. It creates and assigns a `NodeData` to each element in the DOM document `doc` for the given medium `media`. While searching for externally stored CSS style sheets, base URL `base` is used.

There exist several variants of the `assignDOM()` method in the `CSSFactory` class that allow to specify a custom `MatchCondition` for matching pseudoclasses (as discussed in the pseudoclasses section) and/or a custom `NetworkProcessor` for obtaining the imported style sheets as discussed in the parsing section.

Retrieving the Style of DOM Elements

When the analyzing part is done for the style sheet, the computed mapping between DOM elements and the `NodeData` structures representing their styles is available as a `StyleMap` structure. This structure extends the standard Java `Map` structure and the style of a particular DOM element may be therefore obtained using the following method:

- **`NodeData get(org.w3c.dom.Element)`**

The `NodeData` structure provides two basic methods:

- `public <T extends CSSProperty> T getProperty(String name, boolean includeInherited)`, which returns a `CSSProperty`.

Basically, these properties are the implementations of the `CSSProperty` by specialized `enums`. To distinguish between constants values and variable values, following contract is used for the enum values:

- UPPERCASE are the constant values and
- lowercase are the values that contain additional information, which can be retrieved by the following function.
- `public <T extends Term<?>> T getValue(Class<T> clazz, String name, boolean includeInherited);` retrieves a value of type `Term`, determined in package `cz.vutbr.web.css`.

For both these methods, there are equivalent ones defined with automatic inclusion of inherited properties/terms.

Example of enum values for the `CSSProperty` `max-height`: `length`, `percentage`, `NONE`, `INHERIT`

Value `INHERIT` is present for all properties, `length` and `percentage` determine type of token which is about to be retrieved to get exact information about style

For determining the type of the `CSSProperty`, compiler inference is used. Strictly speaking that means that the type of L-value(expression at the left side of equal-sign) is used to determine type to which the result is casted. This could lead in `ClassCastException` in cases when the user uses invalid combination of L-value type and property name.
When there is no L-value, the supertype (that is `CSSProperty`) is used to cast the resulting expression. This is always valid cast.

The following example shows how to obtain the value of a top margin of an element:

```
//get the element style
StyleMap map = CSSFactory.assignDOM(doc,
encoding, base, medium, true);
NodeData style = map.get(element);
//get the type of the assigned value
CSSProperty.Margin mm =
style.getProperty("margin-top");
System.out.println("margin-top=" + mm);
//if a length is specified, obtain the exact
value
if (mm == Margin.length)
{
    TermLength mtop =
style.getValue(TermLength.class, "margin-
top");
    System.out.println("value=" + mtop);
}
```

Obtaining the Style of Pseudo-Elements

CSS specification allows the use of several pseudo-elements for addressing specific parts of the existing DOM elements. The style of the pseudo-elements may be accessed using the following method of the `StyleMap` obtained for the DOM tree:

- **`boolean hasPseudo(org.w3c.dom.Element, Selector.PseudoDeclaration)`** checks whether the given element has some style for the particular pseudo-element declared.
- **`NodeData get(org.w3c.dom.Element, Selector.PseudoDeclaration)`** obtains the style of the particular pseudo-element of the given DOM element.

The pseudo-elements are specified by a `Selector.PseudoDeclaration` pseudo-element value and may be one of the following: `BEFORE`, `AFTER`, `FIRST_LETTER` or `FIRST_LINE`. The remaining values of `Selector.PseudoDeclaration` correspond to pseudo-classes that must be treated differently as described in the following section.

Applying Pseudo-Classes

jStyle parser supports a subset of the available CSS3 pseudo-classes: the structural pseudo-classes and the dynamic pseudo-classes.

The *structural pseudo-classes* (such as `:first-child`) are supported and evaluated automatically. Their defined style is automatically included in the resulting style assigned to the appropriate DOM elements in the resulting `StyleMap`.

The *dynamic pseudo-classes* (such as `:hover`) are more complicated. Any element may belong dynamically to several pseudo-classes that influence the resulting style of the element itself but also the style of its child elements. Therefore, before the DOM style is evaluated as described in previous sections, the current pseudo-classes must be assigned to the individual elements in order to compute the resulting styles properly.

The default behavior of jStyleParser corresponds to the standard static HTML file displaying behavior:

- Links represented using the <a> tags are assigned the `:link` pseudo-class.
- No dynamic pseudo-classes are assigned to the remaining elements.

For specifying other pseudo-classes for different elements, a special `MatchCondition` mechanism may be used. Generally, a `MatchCondition` specifies an additional condition applied when matching specific parts of the CSS selectors. Its default implementation `MatchConditionImpl` implements the default behavior described above. For implementing a better behavior, a configurable `MatchConditionOnElements` implementation is prepared. It allows do assign a set of pseudo-classes directly to given DOM elements or to specified element names. It usage is demonstrated on the following code:

```
//obtain the elements e1 and e2 that should be
assigned the style. e.g.:
Element e1 =
document.getElementById('element1');
Element e2 = ... //or any other way of
obtaining a DOM Element

//Create the match condition. Preserve the
default behavior:
//  all <a> links are assigned the :link class
MatchConditionOnElements cond = new
MatchConditionOnElements("a",
PseudoDeclaration.LINK);

//assign pseudo-classes to the selected
elements
cond.addMatch(e1, PseudoDeclaration.HOVER);
cond.addMatch(e2, PseudoDeclaration.VISITED);

//register the match condition so that it is
```

```
used by jStyleParser
CSSFactory.registerDefaultMatchCondition(cond);

//evaluate the DOM styles as normally
StyleMap decl = CSSFactory.assignDOM(doc,
null, base, "screen", true);
...
```

When the pseudo-class assignment changes, the match condition must be reconfigured and the DOM style must be recomputed.

Alternatively, it is possible to pass the `MatchCondition` directly to the DOM analyzer instead of using the global `registerDefaultMatchCondition()` function. When the Analyzer is used for evaluating the DOM (as discussed here), the match condition may be registered using its `registerMatchCondition` function. When using the similified method (using CSSFactory), the match condition may be passed as an additional argument to the `CSSFactory.assignDOM()` method.

Internal Structure of the Library

The code is divided into following packages:

1. `cz.vutbr.web.css`,
2. `cz.vutbr.web.csskit`,
3. `cz.vutbr.web.domassign` and

4. `test`.

The first package provides in general the contracts used in other applications, the second one contains its implementation. The third package is about assigning CSS rules to HTML elements. The last one contains test units.

Package cz.vutbr.web.css

In addition to several implementation interfaces, this package provides a general entry point of the jStyleParser library. It is the `CSSFactory` class. By default, the interfaces use an implementation provided by `cz.vutbr.csskit` package.

Another remarkable class in this package is the `CSSProperty` interface, which provides a base for CSS properties. By implementing this interface, new CSS properties can be added.

Package cz.vutbr.web.csskit

This package provides a default implementation of `cz.vutbr.web.css`. This can be changed by registering other implementation by calling the appropriate methods of `CSSFactory`.

Internally, it uses ANTLR to parse CSS input into structures defined by contracts in the package `cz.vutbr.web.css`.

Package cz.vutbr.web.domassign

This package provides among others an `Analyzer` class, which is able sort the CSS declarations, to classify them according to a CSS medium and finally, to assign them to the DOM elements.

The transformation core is implemented in the `DeclarationTransformer` class.

Extending Current Version

An extra work was done to simplify the implementation of new CSS standards. The implementation is defined by interface contracts. To implement another CSS parsing level, additional work must be done:

- `SupportedCSS`, which determines the names of the supported CSS properties, and their default values must be replaced with a new implementation. See current implementation in `cz.vutbr.web.domassign.SupportedCSS21` for details. The new implementation must be registered in the CSSFactory.
- If new CSS properties were added, their implementations of `CSSProperty` must be added, preferably by enum. `DeclarationTransformer` must be then informed that there are new properties and their conversion methods must be written.
  Please note that in current implementation, there are

conversion methods defined for all visual CSS 2.1 properties, but they are missing for aural ones, even if `CSSProperty` implementations for aural properties are well defined.

- If the syntax of CSS significantly changes, grammar files `CSS.g` and `CSSTreeParser.g` should be rewritten. In current version, they are written in a way that should simplify future migration to the grammar of CSS 3.0 (as seen at specification draft). Then new parser should be generated.
- If any changes are done in CSS selectors, the `Analyzer` must be made aware of these changes for the classification of DOM elements according to their selectors.

Extending Performance

During the implementation, some additional storage methods have been tested for `NodeData` storage considering time and spatial complexity. Current implementation is a compromise, specialized to lower memory usage. All parts of the library can be changed to use a different implementation by changing the `CSSFactory`. Other factories that can be replaced are factory for creating terms (`TermFactory`) and factory for creating CSS rule parts (`RuleFactory`).