org.htmlparser.lexer

Class Lexer

java.lang.Object
  └ **org.htmlparser.lexer.Lexer**

All Implemented Interfaces:
Serializable, NodeFactory

```
public class Lexer
extends Object
implements Serializable, NodeFactory
```

This class parses the HTML stream into nodes. There are three major types of nodes (lexemes):

- Remark
- Text
- Tag

Each time `nextNode()` is called, another node is returned until the stream is exhausted, and `null` is returned.

See Also:
Serialized Form

Field Summary

protected Cursor	**mCursor** The current position on the page.
protected static int	**mDebugLineTrigger** Line number to trigger on.
protected NodeFactory	**mFactory** The factory for new nodes.
protected Page	**mPage** The page lexemes are retrieved from.
static boolean	**STRICT_REMARKS** Process remarks strictly flag.
static String	**VERSION_DATE** The date of the version ("Jun 10, 2006").
static double	**VERSION_NUMBER** The floating point version number (1.6).

static String	**VERSION_STRING** The display version ("1.6 (Release Build Jun 10, 2006)").
static String	**VERSION_TYPE** The type of version ("Release Build").

Constructor Summary

Lexer() Creates a new instance of a Lexer.
Lexer(Page page) Creates a new instance of a Lexer.
Lexer(String text) Creates a new instance of a Lexer.
Lexer(URLConnection connection) Creates a new instance of a Lexer.

Method Summary

Remark	**createRemarkNode**(Page page, int start, int end) Create a new remark node.
Text	**createStringNode**(Page page, int start, int end) Create a new string node.
Tag	**createTagNode**(Page page, int start, int end, Vector attributes) Create a new tag node.
String	**getCurrentLine**() Get the current line.
int	**getCurrentLineNumber**() Get the current line number.
Cursor	**getCursor**() Get the current scanning position.
NodeFactory	**getNodeFactory**() Get the current node factory.
Page	**getPage**() Get the page this lexer is working on.
int	**getPosition**() Get the current cursor position.
static String	**getVersion**() Return the version string of this parser.
static void	**main**(String[] args) Mainline for command line operation
protected Node	**makeRemark**(int start, int end) Create a remark node based on the current cursor and the one provided.
protected	

Node	**makeString**(int start, int end) Create a string node based on the current cursor and the one provided.
protected Node	**makeTag**(int start, int end, Vector attributes) Create a tag node based on the current cursor and the one provided.
Node	**nextNode**() Get the next node from the source.
Node	**nextNode**(boolean quotesmart) Get the next node from the source.
Node	**parseCDATA**() Return CDATA as a text node.
Node	**parseCDATA**(boolean quotesmart) Return CDATA as a text node.
protected Node	**parseJsp**(int start) Parse a java server page node.
protected Node	**parsePI**(int start) Parse an XML processing instruction.
protected Node	**parseRemark**(int start, boolean quotesmart) Parse a comment.
protected Node	**parseString**(int start, boolean quotesmart) Parse a string node.
protected Node	**parseTag**(int start) Parse a tag.
void	**reset**() Reset the lexer to start parsing from the beginning again.
protected void	**scanJIS**(Cursor cursor) Advance the cursor through a JIS escape sequence.
void	**setCursor**(Cursor cursor) Set the current scanning position.
void	**setNodeFactory**(NodeFactory factory) Set the current node factory.
void	**setPage**(Page page) Set the page this lexer is working on.
void	**setPosition**(int position) Set the current cursor position.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

VERSION_NUMBER

```
public static final double VERSION_NUMBER
```

The floating point version number (1.6).

See Also:
Constant Field Values

VERSION_TYPE

```
public static final String VERSION_TYPE
```

The type of version ("Release Build").

See Also:
Constant Field Values

VERSION_DATE

```
public static final String VERSION_DATE
```

The date of the version ("Jun 10, 2006").

See Also:
Constant Field Values

VERSION_STRING

```
public static final String VERSION_STRING
```

The display version ("1.6 (Release Build Jun 10, 2006)").

See Also:
Constant Field Values

STRICT_REMARKS

```
public static boolean STRICT_REMARKS
```

Process remarks strictly flag. If `true`, remarks are not terminated by ---$gt; or --!$gt;, i.e. more than two dashes. If `false`, a more lax (and closer to typical browser handling) remark parsing is used. Default `true`.

---

mPage

protected Page **mPage**

The page lexemes are retrieved from.

---

mCursor

protected Cursor **mCursor**

The current position on the page.

---

mFactory

protected NodeFactory **mFactory**

The factory for new nodes.

---

mDebugLineTrigger

protected static int **mDebugLineTrigger**

Line number to trigger on. This is tested on each `nextNode()` call, as a debugging aid. Alter this value and set a breakpoint on the guarded statement. Remember, these line numbers are zero based, while most editors are one based.

See Also:
nextNode()

Constructor Detail

Lexer

public **Lexer**()

Creates a new instance of a Lexer.

Lexer

public **Lexer**(Page page)

Creates a new instance of a Lexer.

Parameters:
`page` - The page with HTML text.

Lexer

```
public Lexer(String text)
```

Creates a new instance of a Lexer.

Parameters:
`text` - The text to parse.

Lexer

```
public Lexer(URLConnection connection)
      throws ParserException
```

Creates a new instance of a Lexer.

Parameters:
`connection` - The url to parse.
Throws:
`ParserException` - If an error occurs opening the connection.

Method Detail

getVersion

```
public static String getVersion()
```

Return the version string of this parser.

Returns:
A string of the form:

```
"[floating point number] ([build-type] [build-date])"
```

---

getPage

```
public Page getPage()
```

Get the page this lexer is working on.

Returns:
The page that nodes are being read from.

---

setPage

```
public void setPage(Page page)
```

Set the page this lexer is working on.

Parameters:
`page` - The page that nodes will be read from.

---

getCursor

```
public Cursor getCursor()
```

Get the current scanning position.

Returns:
The lexer's cursor position.

---

setCursor

```
public void setCursor(Cursor cursor)
```

Set the current scanning position.

Parameters:
`cursor` - The lexer's new cursor position.

---

getNodeFactory

```
public NodeFactory getNodeFactory()
```

Get the current node factory.

Returns:
The lexer's node factory.

---

setNodeFactory

```
public void setNodeFactory(NodeFactory factory)
```

Set the current node factory.

Parameters:
`factory` - The node factory to be used by the lexer.

---

getPosition

```
public int getPosition()
```

Get the current cursor position.

Returns:

The current character offset into the source.

---

setPosition

```
public void setPosition(int position)
```

Set the current cursor position.

Parameters:
`position` - The new character offset into the source.

---

getCurrentLineNumber

```
public int getCurrentLineNumber()
```

Get the current line number.

Returns:
The line number the lexer's working on.

---

getCurrentLine

```
public String getCurrentLine()
```

Get the current line.

Returns:
The string the lexer's working on.

---

reset

```
public void reset()
```

Reset the lexer to start parsing from the beginning again. The underlying components are reset such that the next call to `nextNode()` will return the first lexeme on the page.

---

nextNode

```
public Node nextNode()
             throws ParserException
```

Get the next node from the source.

Returns:
A Remark, Text or Tag, or `null` if no more lexemes are present.
Throws:

ParserException - If there is a problem with the underlying page.

---

nextNode

```
public Node nextNode(boolean quotesmart)
              throws ParserException
```

Get the next node from the source.

Parameters:
quotesmart - If true, strings ignore quoted contents.
Returns:
A Remark, Text or Tag, or null if no more lexemes are present.
Throws:
ParserException - If there is a problem with the underlying page.

---

parseCDATA

```
public Node parseCDATA()
                throws ParserException
```

Return CDATA as a text node. According to appendix B.3.2 Specifying non-HTML data of the HTML 4.01 Specification:
Element content
When script or style data is the content of an element (SCRIPT and STYLE), the data begins immediately after the element start tag and ends at the first ETAGO ("</") delimiter followed by a name start character ([a-zA-Z]); note that this may not be the element's end tag. Authors should therefore escape "</" within the content. Escape mechanisms are specific to each scripting or style sheet language.

Returns:
The TextNode of the CDATA or null if none.
Throws:
ParserException - If a problem occurs reading from the source.

---

parseCDATA

```
public Node parseCDATA(boolean quotesmart)
                throws ParserException
```

Return CDATA as a text node. Slightly less rigid than parseCDATA() this method provides for parsing CDATA that may contain quoted strings that have embedded ETAGO ("</") delimiters and skips single and multiline comments.

Parameters:
quotesmart - If true the strict definition of CDATA is extended to allow for single or double quoted ETAGO ("</") sequences.
Returns:

The `TextNode` of the CDATA or `null` if none.

Throws:

`ParserException` - If a problem occurs reading from the source.

See Also:

`parseCDATA()`

---

createStringNode

```
public Text createStringNode(Page page,
                             int start,
                             int end)
```

Create a new string node.

Specified by:

`createStringNode` in interface `NodeFactory`

Parameters:

`page` - The page the node is on.
`start` - The beginning position of the string.
`end` - The ending positiong of the string.

Returns:

The created Text node.

---

createRemarkNode

```
public Remark createRemarkNode(Page page,
                               int start,
                               int end)
```

Create a new remark node.

Specified by:

`createRemarkNode` in interface `NodeFactory`

Parameters:

`page` - The page the node is on.
`start` - The beginning position of the remark.
`end` - The ending positiong of the remark.

Returns:

The created Remark node.

---

createTagNode

```
public Tag createTagNode(Page page,
                         int start,
                         int end,
                         Vector attributes)
```

Create a new tag node. Note that the attributes vector contains at least one element, which is the tag name (standalone attribute) at position zero. This can be used to decide which type of node to create, or gate other processing that may be appropriate.

Specified by:

`createTagNode` in interface `NodeFactory`

Parameters:

`page` - The page the node is on.
`start` - The beginning position of the tag.
`end` - The ending positiong of the tag.
`attributes` - The attributes contained in this tag.

Returns:

The created Tag node.

---

scanJIS

```
protected void scanJIS(Cursor cursor)
                throws ParserException
```

Advance the cursor through a JIS escape sequence.

Parameters:

`cursor` - A cursor positioned within the escape sequence.

Throws:

`ParserException` - If a problem occurs reading from the source.

---

parseString

```
protected Node parseString(int start,
                           boolean quotesmart)
                    throws ParserException
```

Parse a string node. Scan characters until "</", "<%", "<!" or < followed by a letter is encountered, or the input stream is exhausted, in which case `null` is returned.

Parameters:

`start` - The position at which to start scanning.
`quotesmart` - If `true`, strings ignore quoted contents.

Returns:

The parsed node.

Throws:

`ParserException` - If a problem occurs reading from the source.

---

makeString

```
protected Node makeString(int start,
                          int end)
                   throws ParserException
```

Create a string node based on the current cursor and the one provided.

Parameters:
`start` - The starting point of the node.
`end` - The ending point of the node.
Returns:
The new Text node.
Throws:
`ParserException` - If the nodefactory creation of the text node fails.

---

parseTag

```
protected Node parseTag(int start)
                 throws ParserException
```

Parse a tag. Parse the name and attributes from a start tag.

From the HTML 4.01 Specification, W3C Recommendation 24 December 1999
http://www.w3.org/TR/html4/intro/sgmltut.html#h-3.2.2

3.2.2 Attributes

Elements may have associated properties, called attributes, which may have values (by default, or set by authors or scripts). Attribute/value pairs appear before the final ">" of an element's start tag. Any number of (legal) attribute value pairs, separated by spaces, may appear in an element's start tag. They may appear in any order.

In this example, the id attribute is set for an H1 element: `<H1 id="section1">` This is an identified heading thanks to the id attribute `</H1>` By default, SGML requires that all attribute values be delimited using either double quotation marks (ASCII decimal 34) or single quotation marks (ASCII decimal 39). Single quote marks can be included within the attribute value when the value is delimited by double quote marks, and vice versa. Authors may also use numeric character references to represent double quotes (") and single quotes ('). For doublequotes authors can also use the character entity reference ".

In certain cases, authors may specify the value of an attribute without any quotation marks. The attribute value may only contain letters (a-z and A-Z), digits (0-9), hyphens (ASCII decimal 45), periods (ASCII decimal 46), underscores (ASCII decimal 95), and colons (ASCII decimal 58). We recommend using quotation marks even when it is possible to eliminate them.

Attribute names are always case-insensitive.

Attribute values are generally case-insensitive. The definition of each attribute in the reference manual indicates whether its value is case-insensitive.

All the attributes defined by this specification are listed in the attribute index.

This method uses a state machine with the following states:

1. state 0 - outside of any attribute
2. state 1 - within attributre name

3. state 2 - equals hit
4. state 3 - within naked attribute value.
5. state 4 - within single quoted attribute value
6. state 5 - within double quoted attribute value
7. state 6 - whitespaces after attribute name could lead to state 2 (=)or state 0

The starting point for the various components is stored in an array of integers that match the initiation point for the states one-for-one, i.e. bookmarks[0] is where state 0 began, bookmarks[1] is where state 1 began, etc. Attributes are stored in a `Vector` having one slot for each whitespace or attribute/value pair. The first slot is for attribute name (kind of like a standalone attribute).

Parameters:
`start` - The position at which to start scanning.
Returns:
The parsed tag.
Throws:
`ParserException` - If a problem occurs reading from the source.

---

makeTag

```
protected Node makeTag(int start,
                       int end,
                       Vector attributes)
                throws ParserException
```

Create a tag node based on the current cursor and the one provided.

Parameters:
`start` - The starting point of the node.
`end` - The ending point of the node.
`attributes` - The attributes parsed from the tag.
Returns:
The new Tag node.
Throws:
`ParserException` - If the nodefactory creation of the tag node fails.

---

parseRemark

```
protected Node parseRemark(int start,
                           boolean quotesmart)
                    throws ParserException
```

Parse a comment. Parse a remark markup.

From the HTML 4.01 Specification, W3C Recommendation 24 December 1999
http://www.w3.org/TR/html4/intro/sgmltut.html#h-3.2.4

3.2.4 Comments

HTML comments have the following syntax:

```
<!-- this is a comment -->

<!-- and so is this one,

which occupies more than one line -->
```

White space is not permitted between the markup declaration open delimiter("<!") and the comment open delimiter ("--"), but is permitted between the comment close delimiter ("--") and the markup declaration close delimiter (">"). A common error is to include a string of hyphens ("---") within a comment. Authors should avoid putting two or more adjacent hyphens inside comments. Information that appears between comments has no special meaning (e.g., character references are not interpreted as such). Note that comments are markup.

This method uses a state machine with the following states:

1. state 0 - prior to the first open delimiter (first dash)
2. state 1 - prior to the second open delimiter (second dash)
3. state 2 - prior to the first closing delimiter (first dash)
4. state 3 - prior to the second closing delimiter (second dash)
5. state 4 - prior to the terminating >

All comment text (everything excluding the < and >), is included in the remark text. We allow terminators like --!> even though this isn't part of the spec.

Parameters:

`start` - The position at which to start scanning.

`quotesmart` - If `true`, strings ignore quoted contents.

Returns:

The parsed node.

Throws:

`ParserException` - If a problem occurs reading from the source.

---

makeRemark

```
protected Node makeRemark(int start,
                          int end)
                   throws ParserException
```

Create a remark node based on the current cursor and the one provided.

Parameters:

`start` - The starting point of the node.

`end` - The ending point of the node.

Returns:

The new Remark node.

Throws:

`ParserException` - If the nodefactory creation of the remark node fails.

---

parseJsp

```
protected Node parseJsp(int start)
                 throws ParserException
```

Parse a java server page node. Scan characters until "%>" is encountered, or the input stream is exhausted, in which case `null` is returned.

Parameters:
`start` - The position at which to start scanning.
Returns:
The parsed node.
Throws:
`ParserException` - If a problem occurs reading from the source.

---

parsePI

```
protected Node parsePI(int start)
                throws ParserException
```

Parse an XML processing instruction. Scan characters until "?>" is encountered, or the input stream is exhausted, in which case `null` is returned.

Parameters:
`start` - The position at which to start scanning.
Returns:
The parsed node.
Throws:
`ParserException` - If a problem occurs reading from the source.

---

main

```
public static void main(String[] args)
                throws MalformedURLException,
                       ParserException
```

Mainline for command line operation

Parameters:
`args` - [0] The URL to parse.
Throws:
`MalformedURLException` - If the provided URL cannot be resolved.
`ParserException` - If the parse fails.
