Tokenization in *SProUT*

Witold Drożdżyński, Jakub Piskorski
German Research Center for Artificial Intelligence
Stuhsatzenhausweg 3, 66123 Saarbrücken, Germany
{witold,piskorsk}@dfki.de

VERSION APRIL 2005
(SPROUT VERSION 4.0 AND HIGHER)

1 Introduction

Tokenization is commonly seen as an independent process of linguistic analysis, in which the input stream of characters is segmented into an ordered sequence of word-like units, usually called tokens, which function as input items for subsequent steps of linguistic processing. Tokens may correspond to words, numbers, punctuation marks, or even proper names. The recognized tokens are usually classified according to their syntax, but the way in which such classification is done may vary. Since the notion of tokenization seems to have different meanings to different people, some tokenization tools fulfill additional tasks, like for instance isolation of sentences, handling of end-line hyphenations or conjoined clitics and contractions [Greffenstette and Tapanainen 94]. Some software systems for performing tokenization are presented in [Greffenstette et al. 00], [Grover et al. 00], and [Gillam, 99]. The famous general-purpose character stream scanners lex and flex are described in [Levine et al., 92]. In this paper, we present the tokenization tool used in *SProUT* [Becker et al., 02]. Note that it can be used independently of *SProUT*.

2. Tokenization in SProUT

The tokenization in *SProUT* is subdivided into four steps: (1) *segmentation*, (2) *classification*, (3) *postsegmentation*, and (4) *subclassification*. The task of segmentation is to identify *token candidates* by using a list of predefined whitespaces (e.g., blanks, tabs, etc.). Obviously, this technique is not applicable to languages like Japanese and Chinese, where words are not separated by spaces, and a line break can occur anywhere, even in the middle of a word.

Secondly, each token candidate is classified according to a prespecified list of main token types. In contrast to other approaches, the context information is disregarded during token classification, since the goal is to define tokenization as a clear-cut step of linguistic analysis. Strong decomposition of linguistic processing into independent components allows for preserving a high degree of flexibility. As a consequence, and unlike common tokenizers, neither multiword tokens nor simple named entities (such as date and time expressions) are recognized. Sentences are not isolated, since lexical knowledge might be required to do this effectively. Analogously, hyphenated words are not rejoined since the decision for such additional work is not always straightforward and might require more information than just simple lexicon lookup. Consider as an example German compound coordination, where the common part of the compound may be missing, e.g., "Audio- und Videoprodukten" (audio and video products).

In the third step, all tokens which could not be assigned any of the main token types, undergo postsegmentation. This is performed if and only if a token has the following structure:

$$S_0T_1S_1T_2S_2\ldots S_{n-1}T_nS_n$$

where T_i ($i \in \{1,...,n\}$) is a string which belongs to one of the predefined main token classes and S_i ($i \in \{0,...,n\}$) is a separator or a sequence of separators. Furthermore, S_0 and/or S_n can be empty. The list of separators used for postsegmentation is parametrizable, where a separator may consist of more than one symbol. In general there might exist more than one factorization of an unclassified token which matches the structure shown above. In order to avoid ambiguities, the tokenizer finds the left factorization, e.g., if '$$' and '$' are separators then '$$$' would be factorized to '$$' + '$'.

For the sake of clarity, we give some examples. Let us consider the sequence '<http://www.dfki.de>' and let us assume that there is no main token type which covers such constructions. If we assume the characters '<', '>', ':', '/', and '.' to be separators (do not confound them with whitespaces), the postsegmentation would yield following segmentation into tokens:

< http : / / www . dfki . de >

However, if there is no main token type which covers sequences of letters followed by a sequence of digits (e.g., 'abc123'), then the string '(abc123)' could not be postsegmented, since there is no separator between 'abc' and '123'.

Sometimes, postsegmentation of tokens which could not be assigned any of the main token types, might yield undesired effects. This happens frequently with the tokens which end a sentence. For example, lets consider the token 'http://www.dfki.de.' which due to the trailing period can not be classified as an url address. Instead, in the postsegmentation process it will be decomposed into 10 tokens, namely:

http : / / www . dfki . de .

Obviously, this is not what one would expect. The trailing period is responsible for such a segmentation since the whole string does not belong to any particular token class ('unknown' type). However, there is a way of avoiding such segmentation. The tokenizer provides an option, called initial/final separator trimming, which allows for special treatment of unclassified tokens. It can be activated on demand. If the option is active, then each token classified as 'unknown' is segmented as follows: firstly, initial/final separators are isolated (in our example we have only a final separator – the period), and secondly the tokenizer tries to classify the remaining part as usually. This would lead in our case to a segmentation of the input string into 'http://www.dfki.de' and '.' which is intuitively exactly what we wanted to achieve. If the removal of initial/final separators produces a string which still can not be segmented, then normal postsegmentation is performed as described earlier. Using this option turned to come in particular in handy for token segmentation/classification near sentence boundaries.

Finally, depending on the main token class the tokens undergo additional language and/or domain specific classification. For instance, natural numbers could be subclafssified according to the number of digits they consist of (e.g., two-digit numbers, four-digit numbers). The subclassification may return as a result a set of subtypes. For example, for tokens initialized with a capital, we could define following subclasses: city, first-name, contains-only-consonants, and has-noun-ending (which are not necessarily disjunct). The subclassification schedule precisely specifies for each main token type a list of additional subtypes. The early identification of some domain specific token classes, e.g., product names, simplifies the subsequent processing steps like morphological analysis or named-entity recognition. Further, domain and language specific token classes might be used for defining rules or filters for pre-classification of texts or parts of texts (e.g., sentences, paragraphs) as observed in [Giguet, 96]. An elegant and flexible way of processing multilingual texts and in addition texts in different domains means that the tokenization tool can be variably adapted to the needs of the subsequent components.

The tokenizer provides supplementary positional information to each recognized token which might be needful in some context. An example of a possible output structure for the string '2002' is given below.

$$\begin{bmatrix} start : 20 \\ end : 23 \\ type : natural\_number \\ subtype : four\_digit\_natural\_number \end{bmatrix}$$

Current version of the tokenizer supports among other redirecting the output consisting of a sequence of typed feature structures into an XML stream.

3. Configuration

In order to use the tokenizer, following resources must be provided:

- a list of whitespaces,
- a list of separators,
- definition of main token classes,
- definition of token subclasses,
- a list of names containing the main token classes,
- a list of names containing the token subtypes,
- a subclassification schedule,
- and a type hierarchy (this is necessary only for *SProUT*, will be optional in the future)

We now briefly describe how to create these resources. Most of them are constructed by using the Regular Compiler delivered with *SProUT* [Piskorski et al., 02].

3.1. Whitespaces

Whitespaces are represented as a single finite-state automaton which can be constructed with the regular compiler in a straightforward manner. Simply construct a regular expression consisting of a union of whitespaces symbols (e.g., the regular compiler provides unicode tables for accessing hard-to-find symbols). Note that a whitespace symbol may consist of more than one character, e.g., carriage return followed by line feed (\u000A\u000D). Use the following settings in the regular compiler:

```
encoding: unicode
input type: scanner
file | fsm file | file format: bin_java
separated fsm: false
optimization: global
allow ambiguities: false
```

Actually, this resource will be rarely modified.

3.2. Separators

Separators are represented analogously as a single finite-state automaton. It can be constructed in the same way as described in Section 3.1. The regular compiler settings should also be identical. The list of default separators includes the following symbols:

```
.    ,    /    \    -    :    ;    ‘    “
%    !    ?    ~    #    ^    &    *    +
=    |    `    @    )    ]    }    >    (
[    {    <    $    €    £    ¢    ¥    §
?    ¨
```

Analogously to the whitespaces, separators may consist of more than one character.

3.3. Main Token Types

The set of main token types is represented as a weighted finite-state automaton. Each token class is represented by a regular pattern, whereas one has to specify the ID of the pattern and its name. The ID of the pattern represents its priority. If two token classes are not disjoint, then a corresponding token will be assigned a token type which has a higher priority (smaller ID). The regular compiler settings should be identical as specified in section 3.1. The default main token types are listed in the table below.

ID	TYPE	EXAMPLE
1	any_natural_number	123
2	dot	.
3	comma	,
4	slash	/
5	back_slash	\
6	hyphen	-
7	colon	:
8	semicolon	;
9	apostrophe	‘ `
10	quotation	“
11	exclamation_sign	!
12	percentage_tok	%
13	question_mark	?
14	currency_sign	$ € £ ¢ ¥
15	opening_bracket	([{ <
16	closing_bracket	)] } >
17	other_symbol	~ # ^ & * + = | ` @ _ § ¨
18	all_capital_word	CDU
19	lower_case_word	lower
20	first_capital_word	Capital Łódź
21	mixed_word_first_lower	dKK
22	mixed_word_first_capital	GmbH
24	word_with_hyphen_first_capital	Siemens-Sun-Microsoft-AAA
25	word_with_hyphen_first_lower	monday-tuesday
26	word_with_apostrophe_first_capital	Moody's ABC’ ‘AbC
27	word_with_apostrophe_first_lower	ab’cd ‘abc aBC’
28	e_mail_adress	Piskorski@dfki.de
29	url_address	http://www.dfki.de
30	number_word_first_capital	200ABC
31	number_word_first_lower	200abc
32	word_number_first_capital	Windows2000
33	word_number_first_lower	mk2
34	Other	!!! all other strings !!!

The classes 18-22, 24-27 ansd 30-33 allow for using any letters from any language used within the european zone. The classes 24-27 could be easily restricted to exclude certain constructions (e.g., in French, English) by using the diff operator provided in the regular compiler. The names assigned to patterns can be exported by using the following option of the regular compiler:

```
file | export | pattern | macro-names-and-ids
```

3.4. Token Subtypes

Contrary to the main token types, the set of token subtypes is represented as a finite-state automaton without weights. The individual types are identified by using final emissions in the corresponding regular patterns instead of weights. In this way, all ambiguities are preserved. The regular compiler settings differs slightly and should be set as follows:

```
encoding: unicode
input type: scanner
file | fsm file | file format: bin_java
separated fsm: false
optimization: global
allow ambiguities: true
```

The names assigned to patterns can be exported in analogous way as stated in section 3.3.

3.5. Subclassification Schedule

The subclassification schedule is a an ASCII file, where each line has the following format:

MAIN TYPE SUB TYPE

MAIN TYPE is the main token type identifier and SUB TYPE is a subtype identifier. This line specifies that all tokens of type MAIN TYPE are examined in order to determine whether they can be also of type SUB TYPE. Note, that there can be more than one line initialized with the same main token type identifier.

Important

It is important to note that currently *SProUT* itself does not make use of the subclassification feature. This option will be available in the future.

3.5. Tokenizer Configuration in SProUT's IDE

When using the tokenizer within the SProUT grammar development environment, you have to choose the option *Menu -> Tools -> Configure components* which leads you to the configuration window in which you can launch an instance of a tokenizer in the similar way to installing other processing resources. Further, you can activate/deactivate the initial/final separator trimming option by selecting true/false value in the appropriate field. When referring to tokens in your grammars, please use the type *token*, as demonstrated in the following rule example:

```
potential_last_name :/
        ((token & [TYPE first_capital_word, SURFACE #surface])
        | (token & [TYPE mixed_word_first_capital, SURFACE #surface])
        | (token & [TYPE word_with_hyphen_first_capital, SURFACE #surface])
        | (token & [TYPE word_with_apostrophe_first_capital, SURFACE #surface]))
-> aux-name & [SURFACE #surface].
```

The tokenizer can be used independently of *SProUT*. The following piece of code illustrates all indispensable steps which have to be undertaken in order to initialize the tokenizer and to tokenize a text from file via using the class `de.dfki.lt.sprout.runtime.tokenizer.RunTokenizer`.

```
import java.util.*;
import de.dfki.lt.sprout.runtime.tokenizer.*;

// A full path to tokenizer config file
String sConfigFile = "…"

// Text to tokenize
String sInputText = "…"

// Initialize the tokenizer
RunTokenizer runTokenizer = new  RunTokenizer();
int resultInit = runTokenizer.initTokenizer( sConfigFile );
if ( resultInit < 1 ) System.exit( 1 ); // initialization failed

// Tokenize input text
ArrayList result = runTokenizer.tokenize( sInputText );

// Iterate over results
for (int i = 0; i < al.size(); i++) {
  // obtain i-th token
  TToken token = (TToken)al.get( i );
  System.out.println( token );
 }
```

In the example above solely a higher level function for initializing the tokenizer and for applying it to a given input text has been demonstrated. The following example visualizes a more detailed use of the tokenizer.

```
import java.util.*;
import java.text.*;
import java.io.*;
import de.dfki.lt.sprout.runtime.tokenizer.*;

……

static String sXmlOut = "";
static String sFsXmlOut = "";
```

```
String sText;
ArrayList alOutput;

String sConfigFile;
// set a path to config file
sConfigFile = "Config.txt";
String sTextFile = "Sample.txt";

try { Tokenizer pTokenizer;
      ComplexTokenWriter pComplexTokenWriter = new ComplexTokenWriter();

      // Initialize tokenizer from config file
      pTokenizer = new Tokenizer( sConfigFile  );
      System.out.println( "Status: " + pTokenizer.getStatus() );

      // Load text file to tokenize
      FileInputStream pFIS = new FileInputStream( sTextFile );
      byte byA[] = new byte[pFIS.available()];
      pFIS.read( byA, 0, pFIS.available() );
      pFIS.close();
      ByteArrayInputStream pBAIS = new ByteArrayInputStream( byA );

      // Initialize input reader
      AsciiStreamReader pASR = new  AsciiStreamReader( (InputStream)pBAIS );
      // or use the encoding, e.g. UTF-8
      // AsciiStreamReader pASR = new  AsciiStreamReader( (InputStream)pBAIS, "UTF-8" );
      pComplexTokenWriter.setTokenizer( pTokenizer );

      // Tokenizer input structures
      pTokenizer.tokenize( pASR, pComplexTokenWriter );

      // Obtain tokenization results as list (ArrayList) of TToken
      alOutput = pComplexTokenWriter.getResults();

      sText = new String( byA );
      TextHandler pTextHandler = new TextHandler();
      pTextHandler.loadText( sTextFile );

      TokenIterator pTI;

      // Iterate over all tokens
      pTI = new TokenIterator( alOutput );
      pTI.setTextHandler( pTextHandler );
      for( ; pTI.hasNext(); )  System.out.println( pTI.next() );
   } catch( IOException err ) {
```

For more details concerning the tokenizer, please see the Tokenizer API in Appendix A.

References

[Becker et al., 02] M. Becker, W. Drożdżyński, H.U. Krieger, J. Piskorski, U. Schäfer, F. Xu. *SProUT - Shallow Processing with Typed Feature Structures and Unification*. In Proceedings of ICON 2002 - International Conference on NLP, Mumbai, India, 2002.

[Giguet, 96] E. Giguet. *The Stakes of multilinguality: Multilingual text tokenization in Natural Language Diagnosis*. In Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence Workshop "Future issues for Multilingual Text Processing", Cairns, Australia, August 27, 1996.

[Gillam, 99] R. Gillam. *Text Boundary Analysis in Java*. IBM Corp., Web document: http://www.ibm.com/java/education/boundaries/boundaries.html, 1999.

[Greffenstette and Tapanainen, 94] G. Greffenstette, P. Tapanainen. *What is a word, what is a sentence ? problems of tokenization*. In 3rd International Conference on Computational Lexicography (Complex'94), pages 79-87, Budapest. Research Institute for Linguistics Hungarian Academy of Sciences, 1994.

[Greffenstette et al., 00] G. Grefenstette, A. Schiller, and S. Ad t-Mokhtar. *Recognizing Lexical Patterns in Text*. In: F. Van Eynde, D. Gibbon (eds.): Lexicon Development for Speech and Language Processing, Kluwer Academic Publishers, 2000.

[Grover et al., 00] C. Grover, C. Matheson, A. Mikheev and M. Moens. *LT TTT - A Flexible Tokenisation Tool*. In Proceedings of 2nd International Conference on Language Resources and Evaluation (LREC), 2000.

[Levine et al., 92] J. Levine, T. Mason, D. Brown. *Lex & Yacc*. O'Reilly & Associates Inc., 1992.

[Piskorski et al., 02A] J. Piskorski, W. Drożdżyński, F. Xu, O. Scherf. *A Flexible XML-based Regular Compiler for Creation and Converting Linguistic Resources*. In Proceedings of the 3rd International Conference on Language Resources an Evaluation (LREC) 2002, Las Palmas, Spain, 2002.

APPENDIX A - JAVA DOCUMENTATION

---

de.dfki.lt.sprout.runtime.tokenizer
Class Tokenizer

java.lang.Object
└ **de.dfki.lt.sprout.runtime.tokenizer.Tokenizer**

---

public class **Tokenizer**
extends java.lang.Object
This is the main class of tokenizer package. Tokenization is commonly seen as an independent process of linguistic analysis, in which the input stream of characters is segmented into an ordered sequence of word-like units, usually called tokens, which function as input items for subsequent steps of linguistic processing. Tokens may correspond to words, numbers, punctuation marks, or even proper names.
The tokenization is subdivided in four steps: (1) segmentation into tokens, (2) token classification, (3) token postsegmentation (i.e., unclassified tokens are eventually segmented into smaller units, (4) token subclassification. Token classes, separators, whitespaces, and other data are encoded as weighted finite-state automata.

Since:

JDK 1.3

Field Summary

static java.lang.String	**SURFACE_DEF** A constant which specifies the name of the `surface` attribute.
static java.lang.String	**TOKEN_DEF** A constant which specifies the name of the type for representing token information.
static java.lang.String	**TOKENTYPE_DEF** A constant which specifies the name of the type for representing token type information.
static java.lang.String	**TYPE_DEF** A constant which specifies the name of the `type` attribute.

Constructor Summary

Tokenizer(java.lang.String sConfigFile) Constructs a tokenizer with the config file. The configuration file format bases on java configuration property format.
Tokenizer(de.dfki.lt.tools.TConfiguration pParConf) Constructs a Tokenizer with the configuration object

Method Summary

java.util.ArrayList	**CheckConsistency**(de.dfki.lt.tfs.ShUG grammar, java.lang.String sUnknownClassName) This function performs a consistency check in order to test if all feature structure types and attributes used by the tokenizer component are present in the given type hierarchy.
java.lang.String	**getConfigProperty**(java.lang.String sProp) Gets a given property value from the configuration property set of this tokenizer.
boolean	**getInitialFinalSeparatorTrimming**() Gets the initial/final separator trimming flag value
java.util.Hashtable	**getMainTokenClasses**() Gets an information about main token class names associated with their ids.
java.util.ArrayList	**getMainTokenClassNames**() Gets the list of main token class names.
java.util.Hashtable	**getSpecificTokenClasses**() Gets associations of main token class ids and list of token subclass ids.
java.util.Hashtable	**getSpecificTokenClassNames**()

	Gets associations of specific token class ids and their names.
int	**getStatus**() Gets the status of Tokenizer.
TokenIterator	**getTokenIterator**() Gets the TokenIterator associated with this tokenizer.
void	**setInitialFinalSeparatorTrimming**(boolean b) Sets the initial/final separator trimming flag
void	**setTokenIterator**(TokenIterator pTI) Sets TokenIterator for this tokenizer.
void	**tokenize**(StreamReader pSR, TokenWriter pTokenWriter) Tokenizes a document read from StreamReader and stores the results using token writer.

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

Field Detail

TOKEN_DEF

public static final transient java.lang.String **TOKEN_DEF**

A constant which specifies the name of the type for representing token information.

See Also:

Constant Field Values

SURFACE_DEF

public static final transient java.lang.String **SURFACE_DEF**

A constant which specifies the name of the `surface` attribute.

See Also:

Constant Field Values

TYPE_DEF

public static final transient java.lang.String **TYPE_DEF**

A constant which specifies the name of the `type` attribute.

See Also:

Constant Field Values

TOKENTYPE_DEF

public static final transient java.lang.String **TOKENTYPE_DEF**

A constant which specifies the name of the type for representing token type information.

See Also:

Constant Field Values

Constructor Detail

Tokenizer

public **Tokenizer**(java.lang.String sConfigFile)

Constructs a tokenizer with the config file.
The configuration file format bases on java configuration property format. The list of config file entries:
MainTokenClassesFSM= [path to fsm file containing compiled main token classes]
SeparatorClassesFSM= [path to fsm file containing compiled separator classes]
WSClassesFSM= [path to fsm file containing compiled white-space classes]
SubClassesFSM= [path to fsm file containing compiled subclasses]
ScheduleFile= [path to text file containing information about dependences between main token classes and token subclasses]
MainClassNameList= [path to text file containing ids of main token classes associated with their names]
SubClassNameList= [path to text file containing ids of token subclasses associated with their names]
GrammarFile= [name of grm file]
Initial_final_separator token splitting= [true/false], default value=true: the flag for the initial/final separator trimming

Parameters:

`sConfigFile` - - config file name

Tokenizer

public **Tokenizer**(de.dfki.lt.tools.TConfiguration pParConf)

Constructs a Tokenizer with the configuration object

Parameters:

`pParConf` - - configuration structure

See Also:

`Tconfiguration`

Method Detail

setInitialFinalSeparatorTrimming

public void **setInitialFinalSeparatorTrimming**(boolean b)

Sets the initial/final separator trimming flag

---

getInitialFinalSeparatorTrimming

public boolean **getInitialFinalSeparatorTrimming**()

Gets the initial/final separator trimming flag value

---

tokenize

public void **tokenize**(StreamReader pSR,
TokenWriter pTokenWriter)

Tokenizes a document read from StreamReader and stores the results using token writer.

Parameters:

`pSR` - - the input reader

`pTokenWriter` - - the result writer

---

getStatus

public int **getStatus**()

Gets the status of Tokenizer.

Returns:

status value: 0 - not initialized, 1 - initialized, 2 - tokenization finished successfully

---

getMainTokenClasses

public java.util.Hashtable **getMainTokenClasses**()

Gets an information about main token class names associated with their ids.

key - id of a main class

value - class name

CheckConsistency

public java.util.ArrayList **CheckConsistency**(de.dfki.lt.tfs.ShUG grammar,
java.lang.String sUnknownClassName)

This function performs a consistency check in order to test if all feature structure types and attributes used by the tokenizer component are present in the given type hierarchy.

Parameters:

`grammar` - object containing a type hierarchy for testing the consistency

`sUnknownClassName` - - the name of the unknown token class

Returns:

an array list of String containing all inconsistencies. If no incosistencies were found, the array list will be empty.

---

getMainTokenClassNames

public java.util.ArrayList **getMainTokenClassNames**()

Gets the list of main token class names.

---

getConfigProperty

public java.lang.String **getConfigProperty**(java.lang.String sProp)

Gets a given property value from the configuration property set of this tokenizer.

Returns:

a given property value from the configuration property set of this tokenizer.

---

getSpecificTokenClasses

public java.util.Hashtable **getSpecificTokenClasses**()

Gets associations of main token class ids and list of token subclass ids.

Returns:

associations of main token class ids and list of token subclass ids.

---

getSpecificTokenClassNames

public java.util.Hashtable **getSpecificTokenClassNames**()

Gets associations of specific token class ids and their names.

Returns:

associations of specific token class ids and their names.

---

getTokenIterator

public TokenIterator **getTokenIterator**()

Gets the TokenIterator associated with this tokenizer.

Returns:

the TokenIterator associated with this tokenizer.

---

setTokenIterator

public void **setTokenIterator**(TokenIterator pTI)

Sets TokenIterator for this tokenizer.
Parameters:
`pTI` - TokenIterator to be set for this tokenizer.

---

de.dfki.lt.sprout.runtime.tokenizer
Class AbstractTokenReader

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.AbstractTokenReader**
Direct Known Subclasses:
TokenReader

---

public abstract class **AbstractTokenReader**
extends java.lang.Object
AbstractTokenReader represents a general class for token readers.
Since:
JDK 1.3

Constructor Summary

AbstractTokenReader()	Constructs a new, empty AbstractTokenReader

Method Summary

abstract TToken	**getNextToken**() Returns the next token if available.
abstract TToken	**getTokenAt**(int iPos) Gets the token at the specified index.
abstract boolean	**isNextTokenAvailable**() Tests if there is a token available.
abstract void	**setPosition**(int iPos) Sets the read cursor to the specified position in the input sequence.

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

Constructor Detail

AbstractTokenReader

public **AbstractTokenReader**()
Constructs a new, empty AbstractTokenReader

Method Detail

getNextToken

public abstract TToken **getNextToken**()

Returns the next token if available.

Returns:

the next token available or **null** if no more token can be read.

See Also:

`TToken`

---

getTokenAt

public abstract TToken **getTokenAt**(int iPos)

Gets the token at the specified index.

Parameters:

`iPos` - index of element to return.

Returns:

the token at the specified index.

See Also:

`TToken`

---

setPosition

public abstract void **setPosition**(int iPos)

Sets the read cursor to the specified position in the input sequence.

---

isNextTokenAvailable

public abstract boolean **isNextTokenAvailable**()

Tests if there is a token available.

Returns:

`true` if there is at least one more token available.

---

de.dfki.lt.sprout.runtime.tokenizer
Class TokenReader

java.lang.Object
  └ de.dfki.lt.sprout.runtime.tokenizer.AbstractTokenReader
    └ **de.dfki.lt.sprout.runtime.tokenizer.TokenReader**

---

public class **TokenReader**
extends AbstractTokenReader
TokenReader is an implementation of AbstractTokenReader.
Since:
JDK 1.3

Constructor Summary

TokenReader(TokenIterator pTI) Constructs a new TokenReader using TokenIterator as input source	

Method Summary

TToken	**getNextToken**() Returns the next token if available.
TToken	**getTokenAt**(int iPos) Gets the token at the specified index.
boolean	**isNextTokenAvailable**() Tests if there is a token available to read.
void	**setPosition**(int iPos) Sets the read cursor to the specified position in the input sequence.

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

Constructor Detail

TokenReader

public **TokenReader**(TokenIterator pTI)
Constructs a new TokenReader using TokenIterator as input source

Method Detail

getNextToken

public TToken **getNextToken**()
Returns the next token if available. Returns **null** if no token can be read.
Specified by:
getNextToken in class AbstractTokenReader
Returns:
the next token from the current token index.
See Also:
TToken

getTokenAt

public TToken **getTokenAt**(int iPos)

Gets the token at the specified index. The function does not check if the index is valid. If the index is not valid, the exception will be thrown.

Specified by:

getTokenAt in class AbstractTokenReader

Parameters:

`iPos` - an index into this vector.

Returns:

the token at the specified index.

See Also:

`TToken`

---

setPosition

public void **setPosition**(int iPos)

Sets the read cursor to the specified position in the input sequence.

Specified by:

setPosition in class AbstractTokenReader

Parameters:

`iPos` - the new read cursor position

---

isNextTokenAvailable

public boolean **isNextTokenAvailable**()

Tests if there is a token available to read.

Specified by:

isNextTokenAvailable in class AbstractTokenReader

Returns:

`true` if at least one more token can be read.

---

de.dfki.lt.sprout.runtime.tokenizer

Class TokenWriter

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.TokenWriter**

Direct Known Subclasses:

ComplexTokenWriter

---

public abstract class **TokenWriter**
extends java.lang.Object

TokenWriter is general class for token writers. This class provides an interface for storing tokens.

Since:

JDK 1.3

Constructor Summary

TokenWriter()	Constructs a TokenWriter

Method Summary

abstract int	**writeToken**(TToken pToken) Writes a token

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

Constructor Detail

TokenWriter

public **TokenWriter**()

Constructs a TokenWriter

Method Detail

writeToken

public abstract int **writeToken**(TToken pToken)

Writes a token

Parameters:

pToken - token to write

See Also:

TToken

---

de.dfki.lt.sprout.runtime.tokenizer
Class ComplexTokenWriter

java.lang.Object
  └ de.dfki.lt.sprout.runtime.tokenizer.TokenWriter
    └ **de.dfki.lt.sprout.runtime.tokenizer.ComplexTokenWriter**

---

public class **ComplexTokenWriter**
extends TokenWriter

ComplexTokenWriter is an interface that stores the information about tokens generated by tokenizer. The ComplexTokenWriter stores the data internally and allowes to write the full data into the file using several formats (binary, xml, FS-xml).
Since:
JDK 1.3

Constructor Summary

ComplexTokenWriter() Constructs a new, empty ComplexTokenWriter

Method Summary

java.util.ArrayList	**getResults**() Gets the list of all tokens
Tokenizer	**getTokenizer**() Gets the active tokenizer for this object
void	**setTokenizer**(Tokenizer parTok) Sets the active Tokenizer for this object.
int	**writeResultsToFile**(java.lang.String sFileName) Writes results into the file in binary format.
int	**writeResultsToFSxml**(java.lang.String sFileName) Writes results into the file in the xml format.
int	**writeResultsToXML**(java.lang.String sFileName, TextHandler pTextHandler) Writes results into file in xml format. Tokenizer is required.
int	**writeToken**(TToken pToken) Writes a token to the internal structure.

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

Constructor Detail

ComplexTokenWriter

public **ComplexTokenWriter**()
Constructs a new, empty ComplexTokenWriter

Method Detail

writeToken

public int **writeToken**(TToken pToken)

Writes a token to the internal structure.

Specified by:

writeToken in class TokenWriter

Parameters:

`pToken` - a token to write

See Also:

`TToken`

---

getResults

public java.util.ArrayList **getResults**()

Gets the list of all tokens

Returns:

ArrayList of tokens (TToken)

See Also:

`TToken`

---

setTokenizer

public void **setTokenizer**(Tokenizer parTok)

Sets the active Tokenizer for this object. This function tries to set ShUG object using tokenizer settings. The tokenizer is required for storing results.

See Also:

`Tokenizer`

---

getTokenizer

public Tokenizer **getTokenizer**()

Gets the active tokenizer for this object

Returns:

the active tokenizer for this object

See Also:

`Tokenizer`

---

writeResultsToFSxml

public int **writeResultsToFSxml**(java.lang.String sFileName)

Writes results into the file in the xml format. Tokens are stored as feature structures. Tokenizer is required.

Parameters:

`sFileName` - destination file name

Returns:
The result of the operation: 1 - success 0 - no tokenizer defined -1 - an exception has occured -2 - ShUG object not set (reqiured for this function)

---

writeResultsToFile

public int **writeResultsToFile**(java.lang.String sFileName)

Writes results into the file in binary format.
Tokenizer is required.

Parameters:
`sFileName` - the system-dependent filename

Returns:
operation result

---

writeResultsToXML

public int **writeResultsToXML**(java.lang.String sFileName,
TextHandler pTextHandler)

Writes results into file in xml format.
Tokenizer is required.

Parameters:
`sFileName` - the destination file name
`pTextHandler` - the handler to the text document

Returns:
operation result 1 - success 0 - no tokenizer defined -1 - an exception has occured

---

de.dfki.lt.sprout.runtime.tokenizer

Class TToken

java.lang.Object
  └ de.dfki.lt.sprout.runtime.tokenizer.AbstractShallowObject
    └ **de.dfki.lt.sprout.runtime.tokenizer.TToken**

---

public class **TToken**
extends AbstractShallowObject

TToken represents a class for storing token information.

Since:
JDK 1.3

Constructor Summary

TToken`(int iStartPosIn, int iEndPosIn, int iMainClassIDIn)` Constructs a TToken object, sets start and end text position and class id.

Method Summary

int	**getMainClassID**() Gets the id of the main token class for this token.
java.util.Vector	**getSpecificClasses**() Gets the list of specific classes for this token.
void	**setSpecificClasses**(java.util.Vector vSp) Sets the list of specific classes for the token.
java.lang.String	**toString**() Returns a string representation of this object and its values.

Methods inherited from class de.dfki.lt.sprout.runtime.tokenizer.AbstractShallowObject

getEndPos, getStartPos, getText, getTokenCount, getType, setText, setType

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

TToken

public **TToken**(int iStartPosIn,
int iEndPosIn,
int iMainClassIDIn)

Constructs a TToken object, sets start and end text position and class id.

Parameters:

`iStartPosIn` - - start position in text for the token
`iEndPosIn` - - end position in text for the token
`iMainClassIDIn` - - class id of the token

Method Detail

toString

public java.lang.String **toString**()

Returns a string representation of this object and its values.

---

getMainClassID

public int **getMainClassID**()

Gets the id of the main token class for this token.

Returns:

the id of the main token class for this token.

---

getSpecificClasses

public java.util.Vector **getSpecificClasses**()

Gets the list of specific classes for this token.

Returns:

the list of the specific classes for this token.

---

setSpecificClasses

public void **setSpecificClasses**(java.util.Vector vSp)

Sets the list of specific classes for the token.

Parameters:

`vSp` - the list of specific token classes.

---

de.dfki.lt.sprout.runtime.tokenizer
Class TokenIterator

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.TokenIterator**

All Implemented Interfaces:

java.util.Iterator

---

public class **TokenIterator**
extends java.lang.Object
implements java.util.Iterator

TokenIterator is implementation of Iterator. This class provides functionality to iterate over token units generated by tokenizer.

Since:

JDK 1.3

Constructor Summary

TokenIterator`(java.util.ArrayList parV)`	Constructs a TokenIterator using the ArrayList as a list of TToken objects.
TokenIterator`(java.lang.String sFileName)`	Constructs a TokenIterator loading sequence of TToken objects from file.

Method Summary

TToken	**getToken**`(int index)` Gets the token at the specified index.

int	**getTokenCount**() Gets the number of tokens available.
boolean	**hasNext**() Returns `true` if the iterator has more elements.
java.lang.Object	**next**() Returns the next token available.
void	**remove**() Function not implemented (required by for the implementation of Iterator).
void	**reset**() Sets the read pointer to the beginning of the input sequence.
void	**setPosition**(int iPosition) Sets the read cursor to the specified position in the input sequence.
void	**setTextHandler**(TextHandler pTextHandlerPar) Sets the text handler.

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

Constructor Detail

TokenIterator

public **TokenIterator**(java.lang.String sFileName)
throws java.io.IOException

Constructs a TokenIterator loading sequence of TToken objects from file.

Parameters:

`sFileName` - - file name

See Also:

TToken

TokenIterator

public **TokenIterator**(java.util.ArrayList parV)

Constructs a TokenIterator using the ArrayList as a list of TToken objects.

See Also:

TToken

Method Detail

remove

public void **remove**()

Function not implemented (required by for the implementation of Iterator).
Specified by:
`remove` in interface `java.util.Iterator`

---

reset

public void **reset**()
Sets the read pointer to the beginning of the input sequence.

---

getTokenCount

public int **getTokenCount**()
Gets the number of tokens available.
Returns:
the number of tokens available.

---

hasNext

public boolean **hasNext**()
Returns `true` if the iterator has more elements.
Specified by:
`hasNext` in interface `java.util.Iterator`
Returns:
`true` if there is at least one token available.

---

getToken

public TToken **getToken**(int index)
Gets the token at the specified index. The function does not check if the index is valid. If the index is not valid, the exception will be thrown. If the text handler is set for this object, the getToken sets the corresponding text segment (from the start to the end position) for this object. If the text handler is not set for this object, the getToken returns TToken object, which contain only the position information and does not contain the text (null value).
Parameters:
`index` - - an index into this vector.
Returns:
the token at the specified index.
See Also:
`TToken`

---

setPosition

public void **setPosition**(int iPosition)

Sets the read cursor to the specified position in the input sequence.

Parameters:

`iPosition` - the new read cursor position.

---

next

public java.lang.Object **next**()

Returns the next token available.

Specified by:

`next` in interface `java.util.Iterator`

Returns:

the next token available, null if there is no more token available.

---

setTextHandler

public void **setTextHandler**(TextHandler pTextHandlerPar)

Sets the text handler. The text handler stores and manages the text used by tokenizer.

Parameters:

`pTextHandlerPar` - the text handler to be set.

See Also:

`for more information`

---

de.dfki.lt.sprout.runtime.tokenizer

Class TextHandler

java.lang.Object
└ **de.dfki.lt.sprout.runtime.tokenizer.TextHandler**

---

public class **TextHandler**
extends java.lang.Object

Stores and manages the text used by tokenizer. The text handler is used by token iterator to access text segments corresponding to the particular tokens.

Since:

JDK 1.3

See Also:

`(getToken)`

Constructor Summary

TextHandler() Constructs a TextHandler with an empty text.

Method Summary

StreamReader	**getStreamReader**() Creates an implemention of the StreamReader class.
java.lang.String	**getText**() Gets the text stored in this object.
int	**loadText**(java.lang.String sTextFile) Loads a text.
void	**setText**(java.lang.String sT) Sets the text stored in this object to the specified text.
java.lang.String	**substring**(int iStart, int iEnd) Gets the substring of text stored in this object.

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

Constructor Detail

TextHandler

public **TextHandler**()
Constructs a TextHandler with an empty text.

Method Detail

loadText

public int **loadText**(java.lang.String sTextFile)
Loads a text.
Parameters:
sTextFile - - name of the file that containt text to load.
Returns:
the status of the load operation: 1 - if successful; -1 - if the text cannot be loaded

---

setText

public void **setText**(java.lang.String sT)
Sets the text stored in this object to the specified text.

getText

public java.lang.String **getText**()

Gets the text stored in this object.

Returns:

the text stored in this object.

---

substring

public java.lang.String **substring**(int iStart,
int iEnd)

Gets the substring of text stored in this object.

Parameters:

`iStart` - - start position of the substring text.

`iEnd` - - end position of the substring text.

Returns:

the specified substring or **null** if the substring cannot be accessed

---

getStreamReader

public StreamReader **getStreamReader**()

Creates an implemention of the StreamReader class. The stream reader bases of the text stored in this object.

Returns:

the StreamReader that bases of the text stored in this object.

See Also:

`StreamReader`

---

de.dfki.lt.sprout.runtime.tokenizer

Class StreamReader

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.StreamReader**

Direct Known Subclasses:

AsciiStreamReader

---

public abstract class **StreamReader**
extends java.lang.Object

StreamReader is a general class for stream readers.

Since:

JDK 1.3

Constructor Summary

StreamReader() Constructs a StreamReader with empty input source
StreamReader(java.io.InputStream pISt) Constructs a StreamReader using InputStream as input source

Method Summary

abstract char	**getNextChar**() Gets a next char read form the inernal input source.
abstract boolean	**isNextCharAvailable**() Returns `true` if there is a character to read.

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

Constructor Detail

StreamReader

public **StreamReader**(java.io.InputStream pISt)

Constructs a StreamReader using InputStream as input source

Parameters:

`pISt` - the input source for this StreamReader.

StreamReader

public **StreamReader**()

Constructs a StreamReader with empty input source

Method Detail

getNextChar

public abstract char **getNextChar**()

Gets a next char read form the inernal input source.

Returns:

the next char read form the inernal input source or **0** if there is no more character to read.

---

isNextCharAvailable

public abstract boolean **isNextCharAvailable**()

Returns `true` if there is a character to read.

Returns:

`true` if there is at least one character available in the input source

---

de.dfki.lt.sprout.runtime.tokenizer
Class AsciiStreamReader

java.lang.Object
  └ de.dfki.lt.sprout.runtime.tokenizer.StreamReader
    └ **de.dfki.lt.sprout.runtime.tokenizer.AsciiStreamReader**

---

public class **AsciiStreamReader**
extends StreamReader

AsciiStreamReader represents subclass of StreamReader. This class provides an interface for reading characters from the InputStream. The AsciiStreamReader can be used as the input reader for the tokenize function of the tokenizer.

Since:

JDK 1.3

Constructor Summary

AsciiStreamReader(java.io.InputStream pISt) Constructs a AsciiStreamReader using the InputStream as an input source.
AsciiStreamReader(java.io.InputStream pISt, java.lang.String sEncoding) Constructs a new AsciiStreamReader using the InputStream as an input source and having specified encoding

Method Summary

char	**getNextChar**() Gets a next char read form the inernal input source.
boolean	**isNextCharAvailable**() Returns `true` if there is a character to read.

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

Constructor Detail

AsciiStreamReader

public **AsciiStreamReader**(java.io.InputStream pISt)

Constructs a AsciiStreamReader using the InputStream as an input source.

Parameters:

`pISt` - the input source for this AsciiStreamReader.

AsciiStreamReader

public **AsciiStreamReader**(java.io.InputStream pISt,
java.lang.String sEncoding)

Constructs a new AsciiStreamReader using the InputStream as an input source and having specified encoding

Parameters:

`pISt` - the input source for this AsciiStreamReader.

`sEncoding` - the name of a supported charset

Method Detail

getNextChar

public char **getNextChar**()

Gets a next char read form the inernal input source.

Specified by:

getNextChar in class StreamReader

Returns:

the next char read form the inernal input source or **0** if there is no more character to read.

---

isNextCharAvailable

public boolean **isNextCharAvailable**()

Returns `true` if there is a character to read.

Specified by:

isNextCharAvailable in class StreamReader

Returns:

`true` if there is at least one character available in the input source

---

de.dfki.lt.sprout.runtime.tokenizer
Class AbstractShallowObject

java.lang.Object
└ **de.dfki.lt.sprout.runtime.tokenizer.AbstractShallowObject**

Direct Known Subclasses:
TToken

---

public abstract class **AbstractShallowObject**
extends java.lang.Object
AbstractShallowObject represents a general text item (tokens, gazetteer items).
Since:
JDK 1.3

Constructor Summary

AbstractShallowObject()	Constructs a new AbstractShallowObject with a default initial settings

Method Summary

int	**getEndPos**() Gets the end position in text for this AbstractShallowObject.
int	**getStartPos**() Gets the start position in text for this AbstractShallowObject.
java.lang.String	**getText**() Gets the text of this AbstractShallowObject.
int	**getTokenCount**() Obtains the number of items available in this AbstractShallowObject
int	**getType**() Gets the type of this AbstractShallowObject.
void	**setText**(java.lang.String sPar) Sets the text for this AbstractShallowObject to the specified text.
void	**setType**(int iType) Sets the type for this AbstractShallowObject.

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

Constructor Detail

AbstractShallowObject

public **AbstractShallowObject**()
Constructs a new AbstractShallowObject with a default initial settings

Method Detail

getTokenCount

public int **getTokenCount**()

Obtains the number of items available in this AbstractShallowObject

Returns:

the number of items available in this AbstractShallowObject *

---

setText

public void **setText**(java.lang.String sPar)

Sets the text for this AbstractShallowObject to the specified text.

---

getText

public java.lang.String **getText**()

Gets the text of this AbstractShallowObject.

Returns:

the text of this AbstractShallowObject.

---

getType

public int **getType**()

Gets the type of this AbstractShallowObject.

Returns:

the type of this AbstractShallowObject.

---

setType

public void **setType**(int iType)

Sets the type for this AbstractShallowObject.

Parameters:

`iType` - the type of this AbstractShallowObject.

---

getStartPos

public int **getStartPos**()

Gets the start position in text for this AbstractShallowObject.

Returns:

the start position in text for this AbstractShallowObject.

---

getEndPos

public int **getEndPos**()

Gets the end position in text for this AbstractShallowObject.

Returns:

the end position in text for this AbstractShallowObject.

---

de.dfki.lt.sprout.runtime.tokenizer

Class TokenizerApp

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.TokenizerApp**

---

public class **TokenizerApp**
extends java.lang.Object

TokenizerApp is the class that can be started from the command line using parameters. TokenizerApp allows to tokenize documents. Use **TokenizerApp ?** call to see all start parameters.

Since:

JDK 1.3

Constructor Summary

TokenizerApp()

Method Summary

static void	main(java.lang.String[] args)

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

Constructor Detail

TokenizerApp

public **TokenizerApp**()

Method Detail

main

public static void **main**(java.lang.String[] args)

---

de.dfki.lt.sprout.runtime.tokenizer
Class RunTokenizer

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.tokenizer.RunTokenizer**

---

public class **RunTokenizer**
extends java.lang.Object
RunTokenizer is the class providing functionality for initializing and running the tokenizer. RunTokenizer provides initTokenizer and tokenize methods. With this two methods it is possible to initialize the tokenizer and to tokenize (a) text(s).
Title:
Description:
Copyright: DFKI
Company: DFKI

Constructor Summary

RunTokenizer()

Method Summary

int	**initTokenizer**(java.lang.String sConfigFile) Initializes the Tokenizer from the configuration file.
java.util.ArrayList	**tokenize**(java.lang.String sFileName, java.lang.String sEncoding) Tokenizes a text file and returns the list of tokens.

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

Constructor Detail

RunTokenizer

public **RunTokenizer**()

Method Detail

initTokenizer

public int **initTokenizer**(java.lang.String sConfigFile)

Initializes the Tokenizer from the configuration file.

Parameters:

`sConfigFile` - config file name

Returns:

status of this operation: 0 - not initialized, 1 - initialized

See Also:

`Tokenizer( String sConfigFile )`

---

tokenize

public java.util.ArrayList **tokenize**(java.lang.String sFileName,
java.lang.String sEncoding)
throws java.lang.Exception

Tokenizes a text file and returns the list of tokens.

Parameters:

`sFileName` - the input file name

`sEncoding` - the encoding of the input text, null means default encoding

Returns:

the list of tokens (TToken)

Throws:

`java.lang.Exception`
