# Apache Lucene

Apache Lucene is a high-performance, full-featured text search engine library.

Lucene, an indexing and search library, accepts only plain text input.

The Lucene API is divided into several packages:

## org.apache.lucene.analysis

This package provides for the following functionality.

### Parsing

Applications that build their search capabilities upon Lucene may support documents in various formats – HTML, XML, PDF, Word – just to name a few. Lucene does not care about the Parsing of these and other document formats, and it is the responsibility of the application using Lucene to use an appropriate Parser to convert the original format into plain text before passing that plain text to Lucene.

### Tokenization

Plain text passed to Lucene for indexing goes through a process generally called tokenization. Tokenization is the process of breaking input text into small indexing elements – tokens. The way input text is broken into tokens heavily influences how people will then be able to search for that text.

In some cases simply breaking the input text into tokens is not enough – a deeper *Analysis* may be needed. Lucene includes both pre- and post-tokenization analysis facilities.

Pre-tokenization analysis can include (but is not limited to) stripping HTML markup, and transforming or removing text matching arbitrary patterns or sets of fixed strings.

There are many post-tokenization steps that can be done, including (but not limited to):

* [Stemming](http://en.wikipedia.org/wiki/Stemming) – Replacing words with their stems. For instance with English stemming "bikes" is replaced with "bike"; now query "bike" can find both documents containing "bike" and those containing "bikes".
* [Stop Words Filtering](http://en.wikipedia.org/wiki/Stop_words) – Common words like "the", "and" and "a" rarely add any value to a search. Removing them shrinks the index size and increases performance. It may also reduce "noise" and actually improve search quality.
* [Text Normalization](http://en.wikipedia.org/wiki/Text_normalization) – Stripping accents and other character markings can make for better searching.
* [Synonym Expansion](http://en.wikipedia.org/wiki/Synonym) – Adding in synonyms at the same token position as the current word can mean better matching when users search with words in the synonym set.

### Core Analysis

The analysis package provides the mechanism to convert Strings and Readers into tokens that can be indexed by Lucene. There are four main classes in the package from which all analysis processes are derived. These are:

* [Analyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Analyzer.html) – An Analyzer is responsible for building a [TokenStream](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenStream.html" \o "class in org.apache.lucene.analysis) which can be consumed by the indexing and searching processes.
* [CharFilter](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/CharFilter.html) – CharFilter extends [Reader](http://download.oracle.com/javase/1.5.0/docs/api/java/io/Reader.html?is-external=true) to perform pre-tokenization substitutions, deletions, and/or insertions on an input Reader's text, while providing corrected character offsets to account for these modifications. This capability allows highlighting to function over the original text when indexed tokens are created from CharFilter-modified text with offsets that are not the same as those in the original text. Tokenizers' constructors and reset() methods accept a CharFilter. CharFilters may be chained to perform multiple pre-tokenization modifications.
* [Tokenizer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Tokenizer.html) – A Tokenizer is a [TokenStream](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenStream.html" \o "class in org.apache.lucene.analysis) and is responsible for breaking up incoming text into tokens. In most cases, an Analyzer will use a Tokenizer as the first step in the analysis process. However, to modify text prior to tokenization, use a CharStream subclass (see above).
* [TokenFilter](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenFilter.html) – A TokenFilter is also a [TokenStream](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenStream.html" \o "class in org.apache.lucene.analysis) and is responsible for modifying tokens that have been created by the Tokenizer. Common modifications performed by a TokenFilter are: deletion, stemming, synonym injection, and down casing. Not all Analyzers require TokenFilters.

Few differences between a Tokenizer and Analyzer are:

* The [Analyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Analyzer.html) is responsible for the entire task of creating tokens out of the input text, while the [Tokenizer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Tokenizer.html) is only responsible for breaking the input text into tokens. Very likely, tokens created by the [Tokenizer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Tokenizer.html) would be modified or even omitted by the [Analyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Analyzer.html) (via one or more [TokenFilter](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenFilter.html" \o "class in org.apache.lucene.analysis)s) before being returned.
* [Tokenizer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Tokenizer.html) is a [TokenStream](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/TokenStream.html" \o "class in org.apache.lucene.analysis), but [Analyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Analyzer.html) is not.
* [Analyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Analyzer.html) is "field aware", but [Tokenizer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Tokenizer.html) is not.

Lucene Java provides a number of analysis capabilities; the most commonly used one being the [StandardAnalyzer](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/standard/StandardAnalyzer.html" \o "class in org.apache.lucene.analysis.standard).

[PerFieldAnalyzerWrapper](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/PerFieldAnalyzerWrapper.html) – Most Analyzers perform the same operation on all [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html)s. The PerFieldAnalyzerWrapper can be used to associate a different Analyzer with different [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html)s.

In most of the cases, analysis will decide the performance of the application. The more you analyse, the slower the indexing will be.

One of the most common used Tokenizer is WhiteSpaceTokenizer. A WhitespaceTokenizer is a tokenizer that divides text at whitespace. Adjacent sequences of non-Whitespace characters form tokens.

**Invoking the Analyzer**

Applications usually do not invoke analysis – Lucene does it for them:

* At indexing, as a consequence of [addDocument(doc)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "addDocument(org.apache.lucene.document.Document)), the Analyzer in effect for indexing is invoked for each indexed field of the added document.
* At search, a QueryParser may invoke the Analyzer during parsing. Note that for some queries, analysis does not take place, e.g. wildcard queries.

### Indexing Analysis vs. Search Analysis

Selecting the "correct" analyzer is crucial for search quality, and can also affect indexing and search performance. The "correct" analyzer differs between applications. Here are some rules of thumb:

* Beware of over analysis – might hurt indexing performance.
* Start with same analyzer for indexing and search, otherwise searches would not find what they are supposed to.
* In some cases a different analyzer is required for indexing and search:
  + Certain searches require more stop words to be filtered. (I.e. more than those that was filtered at indexing.)
  + Query expansion by synonyms, acronyms, auto spell correction, etc.

### Implementing your own Analyzer

Read this section later, since you might not use it now.

## [org.apache.lucene.document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/package-summary.html)

The document package provides the logical representation of a [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html) for indexing and searching.

The document package provides the user level logical representation of content to be indexed and searched. The package also provides utilities for working with [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html)s and [Fieldable](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html)s.

### Document and Fieldable

**Document**: Documents are the unit of indexing and search. A Document is a set of fields. Each field has a name and a textual value. A field may be [stored](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html#isStored()) with the document, in which case it is returned with search hits on the document. Thus each document should typically contain one or more stored fields which uniquely identify it.

Note that fields which are *not* [stored](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html#isStored()) are *not* available in documents retrieved from the index.

A [**Fieldable**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html)is a logical representation of a user's content that needs to be indexed or stored. [Fieldable](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html)s have a number of properties that tell Lucene how to treat the content (like indexed, tokenized, stored, etc.)

**Field:**  A field is a section of a Document. Each field has two parts, a name and a value. Values may be free text, provided as a String or as a Reader, or they may be atomic keywords, which are not further processed. Such keywords may be used to represent dates, urls, etc. Fields are optionally stored in the index, so that they may be returned with hits on the document.

Field class has following enums which determine the behaviour of a field:

* **Field.Index:** Specifies whether and how a field should be indexed.
  + [**ANALYZED**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.Index.html#ANALYZED)**:** Index the tokens produced by running the field's value through an Analyzer.
  + [**ANALYZED\_NO\_NORMS**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.Index.html#ANALYZED_NO_NORMS): Expert: Index the tokens produced by running the field's value through an Analyzer, and also separately disable the storing of norms.
  + [**NO**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.Index.html#NO)**:** Do not index the field value.
  + [**NOT\_ANALYZED**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.Index.html#NOT_ANALYZED)**:** Index the field's value without using an Analyzer, so it can be searched.
  + [**NOT\_ANALYZED\_NO\_NORMS**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.Index.html#NOT_ANALYZED_NO_NORMS) **:** Expert: Index the field's value without an Analyzer, and also disable the indexing of norms.
* **Field.Store:** Specifies whether and how a field should be stored.
  + **No**: Do not store the field value in the index.
  + **Yes**: Store the original field value in the index.
* **Field.TermVector:** Specifies whether and how a field should have term vectors.
  + **No**: Do not store term vectors.
  + **With\_Offsets**: Store the term vector + Token offset information
  + **With\_Positions**: Store the term vector + token position information
  + **With\_Positions\_Offsets**: Store the term vector + Token position and offset information
  + **Yes**: Store the term vectors of each document.

**Note**: it is common to refer to [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html)s having [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html)s, even though technically they have [Fieldable](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Fieldable.html)s.

### Working with Documents

A [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html) is something created by the user application. It is applications job to create Documents based on the content of the files application will work with (Word, txt, PDF, Excel or any other format.)

The [DateTools](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/DateTools.html" \o "class in org.apache.lucene.document) is a utility class to make dates and times searchable (remember, Lucene only searches text).

[NumericField](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/NumericField.html) is a special helper class to simplify indexing of numeric values (and also dates) for fast range queries with [NumericRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/NumericRangeQuery.html" \o "class in org.apache.lucene.search) (using a special sortable string representation of numeric values).

The [FieldSelector](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/FieldSelector.html" \o "interface in org.apache.lucene.document) class provides a mechanism to tell Lucene how to load Documents from storage. If no FieldSelector is used, all Fieldables on a Document will be loaded. As an example of the FieldSelector usage, consider the common use case of displaying search results on a web page and then having users click through to see the full document. In this scenario, it is often the case that there are many small fields and one or two large fields (containing the contents of the original file). Before the FieldSelector, the full Document had to be loaded, including the large fields, in order to display the results. Now, using the FieldSelector, one can[FieldSelectorResult.LAZY\_LOAD](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/FieldSelectorResult.html#LAZY_LOAD) the large fields, thus only loading the large fields when a user clicks on the actual link to view the original content.

## [org.apache.lucene.index](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/package-summary.html)

The index package basically provides for two important classes: IndexWriter and IndexReader class.

### IndexWriter

An IndexWriter creates and maintains an index.

The create argument to the [constructor](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#IndexWriter(org.apache.lucene.store.Directory, org.apache.lucene.analysis.Analyzer, boolean, org.apache.lucene.index.IndexWriter.MaxFieldLength)) determines whether a new index is created, or whether an existing index is opened. Note that you can open an index with create=true even while readers are using the index. The old readers will continue to search the "point in time" snapshot they had opened, and won't see the newly created index until they re-open. There are also [constructors](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#IndexWriter(org.apache.lucene.store.Directory, org.apache.lucene.analysis.Analyzer, org.apache.lucene.index.IndexWriter.MaxFieldLength)) with no create argument which will create a new index if there is not already an index at the provided path and otherwise open the existing index.

In either case, documents are added with [addDocument](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "addDocument(org.apache.lucene.document.Document)) and removed with [deleteDocuments(Term)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "deleteDocuments(org.apache.lucene.index.Term)) or [deleteDocuments(Query)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "deleteDocuments(org.apache.lucene.search.Query)). A document can be updated with [updateDocument](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "updateDocument(org.apache.lucene.index.Term, org.apache.lucene.document.Document)) (which just deletes and then adds the entire document). When finished adding, deleting and updating documents, [close](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#close()) should be called.

These changes are buffered in memory and periodically flushed to the [Directory](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/Directory.html) (during the above method calls). A flush is triggered when there are enough buffered deletes see [setMaxBufferedDeleteTerms(int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "setMaxBufferedDeleteTerms(int))) or enough added documents since the last flush, whichever is sooner. For the added documents, flushing is triggered either by RAM usage of the documents (see [setRAMBufferSizeMB(double)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "setRAMBufferSizeMB(double))) or the number of added documents. The default is to flush when RAM usage hits 16 MB. For best indexing speed you should flush by RAM usage with a large RAM buffer. Note that flushing just moves the internal buffered state in IndexWriter into the index, but these changes are not visible to IndexReader until either [commit()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#commit()) or [close()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#close()) is called. A flush may also trigger one or more segment merges which by default run with a background thread so as not to block the addDocument calls (see [below](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html#mergePolicy) for changing the [MergeScheduler](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergeScheduler.html" \o "class in org.apache.lucene.index)).

Opening an IndexWriter creates a lock file for the directory in use. Trying to open another IndexWriter on the same directory will lead to a [LockObtainFailedException](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/LockObtainFailedException.html" \o "class in org.apache.lucene.store). The [LockObtainFailedException](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/LockObtainFailedException.html" \o "class in org.apache.lucene.store) is also thrown if an IndexReader on the same directory is used to delete documents from the index.

Expert: IndexWriter allows an optional [IndexDeletionPolicy](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexDeletionPolicy.html" \o "interface in org.apache.lucene.index) implementation to be specified. You can use this to control when prior commits are deleted from the index. The default policy is[KeepOnlyLastCommitDeletionPolicy](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/KeepOnlyLastCommitDeletionPolicy.html) which removes all prior commits as soon as a new commit is done (this matches behavior before 2.2). Creating your own policy can allow you to explicitly keep previous "point in time" commits alive in the index for some time, to allow readers to refresh to the new commit without having the old commit deleted out from under them. This is necessary on filesystems like NFS that do not support "delete on last close" semantics, which Lucene's "point in time" search normally relies on.

Expert: IndexWriter allows you to separately change the [MergePolicy](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergePolicy.html" \o "class in org.apache.lucene.index) and the [MergeScheduler](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergeScheduler.html" \o "class in org.apache.lucene.index). The [MergePolicy](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergePolicy.html" \o "class in org.apache.lucene.index) is invoked whenever there are changes to the segments in the index. Its role is to select which merges to do, if any, and return a [MergePolicy.MergeSpecification](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergePolicy.MergeSpecification.html" \o "class in org.apache.lucene.index) describing the merges. The default is [LogByteSizeMergePolicy](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/LogByteSizeMergePolicy.html" \o "class in org.apache.lucene.index). Then, the [MergeScheduler](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/MergeScheduler.html" \o "class in org.apache.lucene.index) is invoked with the requested merges and it decides when and how to run the merges. The default is [ConcurrentMergeScheduler](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/ConcurrentMergeScheduler.html" \o "class in org.apache.lucene.index).

### IndexReader

IndexReader is an abstract class, providing an interface for accessing an index. Search of an index is done entirely through this abstract interface, so that any subclass which implements it is searchable.

Concrete subclasses of IndexReader are usually constructed with a call to one of the static open() methods, e.g. [open(Directory, boolean)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexReader.html#open(org.apache.lucene.store.Directory, boolean)).

For efficiency, in this API documents are often referred to via *document numbers*, non-negative integers which each name a unique document in the index. These document numbers are ephemeral--they may change as documents are added to and deleted from an index. Clients should thus not rely on a given document having the same number between sessions.

An IndexReader can be opened on a directory for which an IndexWriter is opened already, but it cannot be used to delete documents from the index then.

### TermFrequencyVector

TermFrequencyVector provides access to stored term vector of a document field. The vector consists of the name of the field, an array of the terms that occur in the field of the Document and a parallel array of frequencies.

We can obtain an instance of TermFrequencyVector by using the IndexReader.getTermFreqVector(int docId,String field) method. Once we have access to this object, we can get all the Terms for a document and frequencies for all the terms using appropriate methods.

**Note:** TermFrequencyVector will have Terms and frequencies only for the fields that had the store TermFrequencyVector field set to true while creation. In cases when the number of documents will be very high, this can decrease the time taken to create the index.

**Note:** TermFrequencyVector reads directly from the .frq file depending on the document ID passed to it.

## org.apache.lucene.search

### Search

Search over indices. Applications usually call [Searcher.search(Query,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Searcher.html#search(org.apache.lucene.search.Query, int)) or [Searcher.search(Query,Filter,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Searcher.html#search(org.apache.lucene.search.Query, org.apache.lucene.search.Filter, int)).

### Query Classes

#### [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html)

Of the various implementations of [Query](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html), the [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) is the easiest to understand and the most often used in applications. A [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) matches all the documents that contain the specified [Term](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/Term.html), which is a word that occurs in a certain [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html). Thus, a [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) identifies and scores all [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html)s that have a [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html) with the specified string in it. Constructing a [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) is as simple as:

TermQuery tq = new TermQuery(new Term("fieldName", "term"));

In this example, the [Query](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html) identifies all [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html)s that have the [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html) named "fieldName" containing the word "term".

#### [BooleanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html)

 A [BooleanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html) contains multiple [BooleanClause](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanClause.html)s, where each clause contains a sub-query ([Query](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html) instance) and an operator (from [BooleanClause.Occur](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanClause.Occur.html)) describing how that sub-query is combined with the other clauses.

The various Clauses are:

1. SHOULD — Use this operator when a clause can occur in the result set, but is not required. If a query is made up of all SHOULD clauses, then every document in the result set matches at least one of these clauses.
2. MUST — Use this operator when a clause is required to occur in the result set. Every document in the result set will match all such clauses.
3. MUST NOT — Use this operator when a clause must not occur in the result set. No document in the result set will match any such clauses.

Boolean queries are constructed by adding two or more [BooleanClause](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanClause.html) instances. If too many clauses are added, a [TooManyClauses](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.TooManyClauses.html) exception will be thrown during searching. This most often occurs when a [Query](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html) is rewritten into a [BooleanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html) with many [TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) clauses, for example by [WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html). The default setting for the maximum number of clauses 1024, but this can be changed via the static method [setMaxClauseCount](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html" \l "setMaxClauseCount(int)) in[BooleanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html).

#### Phrases

Another common search is to find documents containing certain phrases. This is handled two different ways:

1. [**PhraseQuery**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/PhraseQuery.html) — Matches a sequence of [Terms](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/Term.html). [PhraseQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/PhraseQuery.html) uses a slop factor to determine how many positions may occur between any two terms in the phrase and still be considered a match.
2. [**SpanNearQuery**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/spans/SpanNearQuery.html) — Matches a sequence of other [SpanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/spans/SpanQuery.html) instances. [SpanNearQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/spans/SpanNearQuery.html) allows for much more complicated phrase queries since it is constructed from other [SpanQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/spans/SpanQuery.html) instances, instead of only[TermQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html) instances.

#### [TermRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermRangeQuery.html)

The [TermRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermRangeQuery.html) matches all documents that occur in the exclusive range of a lower [Term](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/Term.html) and an upper [Term](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/Term.html), according to [String.compareTo(String)](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/String.html?is-external=true" \l "compareTo(java.lang.String)" \o "class or interface in java.lang). It is not intended for numerical ranges, use[NumericRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/NumericRangeQuery.html) instead. For example, one could find all documents that have terms beginning with the letters a through c. This type of [Query](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html) is frequently used to find documents that occur in a specific date range.

#### [NumericRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/NumericRangeQuery.html)

The [NumericRangeQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/NumericRangeQuery.html) matches all documents that occur in a numeric range. For NumericRangeQuery to work, you must index the values using a special [NumericField](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/NumericField.html).

#### [PrefixQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/PrefixQuery.html), [WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html)

While the [PrefixQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/PrefixQuery.html) has a different implementation, it is essentially a special case of the [WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html). The [PrefixQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/PrefixQuery.html) allows an application to identify all documents with terms that begin with a certain string. The[WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html) generalizes this by allowing for the use of \* (matches 0 or more characters) and ? (matches exactly one character) wildcards.

Note that the [WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html) can be quite slow. Also note that [WildcardQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/WildcardQuery.html) should not start with \* and ?, as these are extremely slow.

#### [FuzzyQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/FuzzyQuery.html)

A [FuzzyQuery](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/FuzzyQuery.html) matches documents that contain terms similar to the specified term. Similarity is determined using [Levenshtein (edit) distance](http://en.wikipedia.org/wiki/Levenshtein). This type of query can be useful when accounting for spelling variations in the collection.

### Changing Similarity

Read later

### Changing Scoring — Expert Level

Read later

### IndexSearcher

Implements search over a single IndexReader.

Applications usually need only call the inherited [search(Query,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/IndexSearcher.html#search(org.apache.lucene.search.Query, int)) or [search(Query,Filter,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/IndexSearcher.html#search(org.apache.lucene.search.Query, org.apache.lucene.search.Filter, int)) methods. For performance reasons, if your index is unchanging, you should share a single IndexSearcher instance across multiple searches instead of creating a new one per-search. If your index has changed and you wish to see the changes reflected in searching, you should use[IndexReader.openIfChanged(org.apache.lucene.index.IndexReader)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexReader.html#openIfChanged(org.apache.lucene.index.IndexReader)) to obtain a new reader and then create a new IndexSearcher from that.

### TopDocs

Represents hits returned by [Searcher.search(Query,Filter,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Searcher.html#search(org.apache.lucene.search.Query, org.apache.lucene.search.Filter, int)) and [Searcher.search(Query,int)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Searcher.html#search(org.apache.lucene.search.Query, int)).

## org.apache.lucene.queryParser

### CharStream

This interface describes a character stream that maintains line and column number positions of the characters. It also has the capability to back up the stream to some extent.

### FastCharStream

This does not do line-number counting, but instead keeps track of the character position of the token in the input, as required by Lucene's [Token](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/analysis/Token.html) API.

### QueryParser

The most important method of this class is [parse(String)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/queryParser/QueryParser.html#parse(java.lang.String)).

The syntax for query strings is as follows:

A Query is a series of clauses. A clause may be prefixed by:

* plus (+) or a minus (-) sign, indicating that the clause is required or prohibited respectively; or
* a term followed by a colon, indicating the field to be searched. This enables one to construct queries which search multiple fields.

A clause may be either:

* a term, indicating all the documents that contain this term; or
* a nested query, enclosed in parentheses. Note that this may be used with a +/- prefix to require any of a set of terms.

### MultiFieldQueryParser

A QueryParser which constructs queries to search multiple fields.

### Token

Describes the input token stream.

### QueryParser.Operator

The default operator for parsing queries. Use [QueryParser.setDefaultOperator(org.apache.lucene.queryParser.QueryParser.Operator)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/queryParser/QueryParser.html#setDefaultOperator(org.apache.lucene.queryParser.QueryParser.Operator)) to change it.

## [org.apache.lucene.store](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/package-summary.html)

This package defines an abstract class for storing persistent data, the Directory, which is a collection of named files written by an IndexOutput and read by an IndexInput. Multiple implementations are provided, including FSDirectory, which uses a file system directory to store files, and RAMDirectory which implements files as memory-resident data structures.

#### Directory

A Directory is a flat list of files. Files may be written once, when they are created. Once a file is created it may only be opened for read, or deleted. Random access is permitted both when reading and writing.

Java's i/o APIs not used directly, but rather all i/o is through this API. This permits things such as:

* implementation of RAM-based indices;
* implementation indices stored in a database, via JDBC;
* implementation of an index as a single file;

There are 2 types of directories:

#### RAMDirectory

A memory-resident [Directory](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/Directory.html) implementation.

This class is not intended to work with huge indexes. Everything beyond several hundred megabytes will waste resources (GC cycles), because it uses an internal buffer size of 1024 bytes, producing millions ofbyte[1024] arrays. This class is optimized for small memory-resident indexes. It also has bad concurrency on multithreaded environments.

#### FSDirectory

Base class for Directory implementations that store index files in the file system. There are currently three core subclasses:

* **[SimpleFSDirectory](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/SimpleFSDirectory.html" \o "class in org.apache.lucene.store)** is a straightforward implementation using java.io.RandomAccessFile. However, it has poor concurrent performance (multiple threads will bottleneck) as it synchronizes when multiple threads read from the same file.
* **NIOFSDirectory** uses java.nio's FileChannel's positional io when reading to avoid synchronization when reading from the same file. Unfortunately, due to a Windows-only Sun JRE bug this is a poor choice for Windows, but on all other platforms this is the preferred choice. Applications using Thread.interrupt() or Future.cancel(boolean) should use SimpleFSDirectory instead. See NIOFSDirectory java doc for details.
* [**MMapDirectory**](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/MMapDirectory.html) uses memory-mapped IO when reading. This is a good choice if you have plenty of virtual memory relative to your index size, eg if you are running on a 64 bit JRE, or you are running on a 32 bit JRE but your index sizes are small enough to fit into the virtual memory space. Java has currently the limitation of not being able to unmap files from user code. The files are unmapped, when GC releases the byte buffers. Due to [this bug](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4724038) in Sun's JRE, MMapDirectory's [IndexInput.close()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/IndexInput.html" \l "close()) is unable to close the underlying OS file handle. Only when GC finally collects the underlying objects, which could be quite some time later, will the file handle be closed. This will consume additional transient disk usage: on Windows, attempts to delete or overwrite the files will result in an exception; on other platforms, which typically have a "delete on last close" semantics, while such operations will succeed, the bytes are still consuming space on disk. For many applications this limitation is not a problem (e.g. if you have plenty of disk space, and you don't rely on overwriting files on Windows) but it's still an important limitation to be aware of. This class supplies a (possibly dangerous) workaround mentioned in the bug report, which may fail on non-Sun JVMs. Applications using[Thread.interrupt()](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html?is-external=true#interrupt()) or [Future.cancel(boolean)](http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Future.html?is-external=true" \l "cancel(boolean)" \o "class or interface in java.util.concurrent) should use [SimpleFSDirectory](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/SimpleFSDirectory.html" \o "class in org.apache.lucene.store) instead. See [MMapDirectory](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/MMapDirectory.html" \o "class in org.apache.lucene.store) java doc for details.

Unfortunately, because of system peculiarities, there is no single overall best implementation. Therefore, we've added the [open(java.io.File)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/FSDirectory.html#open(java.io.File)) method, to allow Lucene to choose the best FSDirectory implementation given your environment, and the known limitations of each implementation. For users who have no reason to prefer a specific implementation, it's best to simply use [open(java.io.File)](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/store/FSDirectory.html#open(java.io.File)). For all others, you should instantiate the desired implementation directly.

## org.apache.lucene.util

Read later

## How to use Lucene in your application

To use Lucene, an application should:

* Create [Document](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html)s by adding [Field](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html)s;
* Create an [IndexWriter](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html) and add documents to it with [addDocument()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/IndexWriter.html" \l "addDocument(org.apache.lucene.document.Document));
* Call [QueryParser.parse()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/queryParser/QueryParser.html" \l "parse(java.lang.String)) to build a query from a string; and
* Create an [IndexSearcher](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/IndexSearcher.html) and pass the query to its [search()](http://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Searcher.html#search(org.apache.lucene.search.Query)) method.

## Different Types of Analyzers

### StopAnalyzer

StopAnalyzer uses **LetterTokenizer** to tokenize the stream and create token stream and **LowerCaseFilter** and **StopFilter** to filter the created tokens.

### WhitespaceAnalyzer

WhiteSpaceAnalyzer uses **WhiteSpaceTokenizer** to tokenize the stream and create token stream.

### ClassicAnalyzer

ClassicAnalyzer uses ClassicTokenizer to tokenize the stream and create token stream. The tokens are then filtered using ClassicFilter, LowerCaseFilter and StopFilter.

## Different Types of Tokenizers

### LetterTokenizer

A LetterTokenizer is a tokenizer that divides text at non-letters. It defines tokens as maximal strings of adjacent letters, as defined by **java.lang.Character.isLetter()** predicate.

**NOTE:** Numerical literals are not considered as letters, hence all the classes that will use LetterTokenizer will break the tokens at boundaries of letters and numeric value.

### WhiteSpaceTokenizer

A WhitespaceTokenizer is a tokenizer that divides text at whitespace. Adjacent sequences of non-Whitespace characters form tokens.

### ClassicTokenizer

A grammar-based tokenizer constructed with JFlex. This is a good Tokenizer to be used with European documents. Salient features of this Tokenizer are:

* Splits words at punctuation characters, removing punctuation. However, a dot that's not followed by whitespace is considered part of a token.
* Splits words at hyphens, unless there's a number in the token, in which case the whole token is interpreted as a product number and is not split.
* Recognizes email addresses and internet hostnames as one token.

## Different Types of Filters

### LowerCaseFilter

LowerCaseFilter normalizes token text to lower case.

### StopFilter

StopFilter Removes stop words from a token stream.