

Lucene Tutorial

Simply Easy Learning



LUCENE TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Lucene Tutorial

Lucene is an open source java based search library. Lucene is very popular and fast search library used in java based application to add document search capability to any kind of application in a very simple and efficient way.

This tutorial will give you great understanding on Lucene concepts needed to understand the complexity of search requirements in enterprise level applications and need of lucene search engine.

Audience

This tutorial is designed for Software Professionals who are willing to learn Lucene search engine Programming in simple and easy steps. This tutorial will give you great understanding on Lucene concepts and after completing this tutorial you will be at intermediate level of expertise from where you can take yourself at higher level of expertise.

Prerequisites

Before proceeding with this tutorial you should have a basic understanding of Java programming language, text editor and execution of programs etc.

Copyright & Disclaimer Notice

©All the content and graphics on this tutorial are the property of tutorialspoint.com. Any content from tutorialspoint.com or this tutorial may not be redistributed or reproduced in any way, shape, or form without the written permission of tutorialspoint.com. Failure to do so is a violation of copyright laws.

This tutorial may contain inaccuracies or errors and tutorialspoint provides no guarantee regarding the accuracy of the site or its contents including this tutorial. If you discover that the tutorialspoint.com site or this tutorial content contains some errors, please contact us at webmaster@tutorialspoint.com

Table of Content

Lucene Tutorial.....	2
Audience.....	2
Prerequisites.....	2
Copyright & Disclaimer Notice	2
Lucene - Overview.....	11
How Search Application works?	11
Lucene's role in search application.....	12
Lucene – First Application	13
Step 2 - Add Required Libraries:	15
Step 3 - Create Source Files:	16
Step 4 - Data & Index directory creation	20
Step 5 - Running the Program:	20
Lucene – Indexing Classes	21
Indexing Classes:	22
Lucene – IndexWriter	23
Introduction.....	23
Class declaration	23
Field.....	23
Class constructors	24
Class methods.....	24
Methods inherited	29
Lucene – Directory	30
Introduction.....	30
Class declaration	30
Field.....	30
Class constructors	30
Class methods.....	30
Methods inherited	32
Lucene – Analyzer.....	33
Introduction.....	33
Class declaration	33
Class constructors	33
Class methods.....	33
Methods inherited	34
Lucene – Document	35
Introduction.....	35
Class declaration	35

Class constructors	35
Class methods.....	35
Methods inherited	36
Lucene – Field	37
Introduction.....	37
Class declaration	37
Class constructors	37
Class methods.....	38
Methods inherited	38
Lucene – Searching Classes.....	39
Searching Classes:.....	39
Lucene – IndexSearcher	40
Introduction.....	40
Class declaration	40
Field.....	40
Class constructors	40
Class methods.....	41
Methods inherited	42
Lucene – Term	43
Introduction.....	43
Class declaration	43
Class constructors	43
Class methods.....	43
Methods inherited	44
Lucene – Query	45
Introduction.....	45
Class declaration	45
Class constructors	45
Class methods.....	45
Methods inherited	46
Lucene – TermQuery	47
Introduction.....	47
Class declaration	47
Class constructors	47
Class methods.....	47
Methods inherited	48
Lucene – TopDocs	49
Introduction.....	49
Class declaration	49

Field.....	49
Class constructors	49
Class methods.....	49
Methods inherited	50
Lucene – Indexing Process	51
Create a document	51
Create a IndexWriter	52
Start Indexing process	52
Example Application	53
Data & Index directory creation	56
Running the Program:	56
Lucene – Indexing Operations	58
Indexing Operations:	58
Lucene – Add Document.....	59
Add a document to an index:	59
Create a IndexWriter	60
Add document and start Indexing process	60
Example Application	60
Data & Index directory creation:	63
Running the Program:	63
Lucene – Update Document.....	65
Update a document to an index.....	65
Create a IndexWriter	65
Update document and start reindexing process	66
Example Application	66
Data & Index directory creation	69
Running the Program:	69
Lucene – Delete Document.....	70
Delete a document from an index.....	70
Create a IndexWriter	70
Delete document and start reindexing process.....	71
Example Application	71
Data & Index directory creation	74
Running the Program:	74
Lucene – Field Options	75
Various Field Options	75
Use of Field Options	75
Example Application	76
Data & Index directory creation	79

Running the Program:	79
Lucene – Search Operations.....	80
Create a QueryParser	81
Create a IndexSearcher	81
Make search	81
Get the document	81
Close IndexSearcher	82
Example Application	82
Data & Index directory creation	84
Running the Program:	84
Lucene – Query Programming	85
Lucene – TermQuery	86
Introduction.....	86
Class declaration	86
Class constructors	86
Class methods.....	86
Methods inherited	87
Usage	87
Example Application	87
Data & Index directory creation	89
Running the Program:	90
Lucene – TermRangeQuery.....	91
Introduction.....	91
Class declaration	91
Class constructors	91
Class methods.....	91
Methods inherited	92
Usage	92
Example Application	92
Data & Index directory creation	95
Running the Program:	95
Lucene – PrefixQuery.....	96
Introduction.....	96
Class declaration	96
Class constructors	96
Class methods.....	96
Methods inherited	96
Usage	97
Example Application	97

Data & Index directory creation	99
Running the Program:	99
Lucene – BooleanQuery.....	101
Introduction.....	101
Class declaration	101
Fields.....	101
Class constructors	101
Class methods.....	101
Methods inherited	102
Usage	102
Example Application	103
Data & Index directory creation	105
Running the Program:	105
Lucene – PhraseQuery	107
Introduction.....	107
Class declaration	107
Class constructors	107
Class methods.....	107
Methods inherited	108
Usage	108
Example Application	108
Data & Index directory creation	111
Running the Program:	111
Lucene – WildcardQuery.....	112
Introduction.....	112
Class declaration	112
Fields.....	112
Class constructors	112
Class methods.....	112
Methods inherited	112
Usage	113
Example Application	113
Data & Index directory creation	115
Running the Program:	115
Lucene – FuzzyQuery	117
Introduction.....	117
Class declaration	117
Fields.....	117
Class constructors	117

Class methods.....	117
Methods inherited	118
Usage	118
Example Application	118
Data & Index directory creation	121
Running the Program:	121
Lucene – MatchAllDocsQuery.....	122
Introduction.....	122
Class declaration	122
Class constructors	122
Class methods.....	122
Methods inherited	122
Usage	123
Example Application	123
Data & Index directory creation	125
Running the Program:	125
Lucene – Analysis	127
Lucene – Token.....	128
Introduction.....	128
Class declaration	128
Fields.....	128
Class constructors	128
Class methods.....	129
Methods inherited	130
Lucene – TokenStream	131
Introduction.....	131
Class declaration	131
Class constructors	131
Class methods.....	131
Methods inherited	132
Lucene – Analyzer.....	133
Introduction.....	133
Class declaration	133
Class constructors	133
Class methods.....	133
Methods inherited	134
Lucene – WhitespaceAnalyzer.....	135
Introduction.....	135
Class declaration	135

Fields	135
Class constructors	135
Class methods	135
Methods inherited	136
Usage	136
Example Application	136
Running the Program:	137
Lucene – SimpleAnalyzer	138
Introduction	138
Class declaration	138
Class constructors	138
Class methods	138
Methods inherited	138
Usage	139
Example Application	139
Running the Program:	140
Lucene – StopAnalyzer	141
Introduction	141
Class declaration	141
Fields	141
Class constructors	141
Class methods	141
Methods inherited	141
Usage	142
Example Application	142
Running the Program:	143
Lucene – StandardAnalyzer	144
Introduction	144
Class declaration	144
Fields	144
Class constructors	144
Class methods	144
Methods inherited	145
Usage	145
Example Application	145
Running the Program:	146
Lucene – Sorting	147
Introduction	147
Sorting By Relevance	147

Sorting By IndexOrder	147
Example Application	148
Data & Index directory creation	151
Running the Program:	151

Lucene - Overview

Lucene is simple yet powerful java based search library. It can be used in any application to add search capability to it. Lucene is open-source project. It is scalable and high-performance library used to index and search virtually any kind of text. Lucene library provides the core operations which are required by any search application. Indexing and Searching.

How Search Application works?

Any search application does the few or all of the following operations.

Step	Title	Description
1	<i>Acquire Raw Content</i>	First step of any search application is to collect the target contents on which search are to be conducted.
2	<i>Build the document</i>	Next step is to build the document(s) from the raw contents which search application can understands and interpret easily.
3	<i>Analyze the document</i>	Before indexing process to start, the document is to be analyzed as which part of the text is a candidate to be indexed. This process is called analyzing the document.
4	<i>Indexing the document</i>	Once documents are built and analyzed, next step is to index them so that this document can be retrived based on certain keys instead of whole contents of the document. Indexing process is similar to indexes in the end of a book where common words are shown with their page numbers so that these words can be tracked quickly instead of searching the complete book.
5	<i>User Interface for Search</i>	Once a database of indexes is ready then application can make any search. To facilitate user to make a search, application must provide a user a mean or u0ser interface where a user can enter text and start the search process.
6	<i>Build Query</i>	Once user made a request to search a text, application should prepare a Query object using that text which can be used to inquire index database to get the relevant details.
7	<i>Search Query</i>	Using query object, index database is then checked to get the relevant details and the content documents.
8	<i>Render Results</i>	Once result is received the application should decide how to show the results to the user using User Interface. How much information is to be shown at first look and so.

Apart from these basic operations, search application can also provide administration user interface providing administrators of the application to control the level of search based on the user profiles. Analytics of search result is another important and advanced aspect of any search application.

Lucene's role in search application

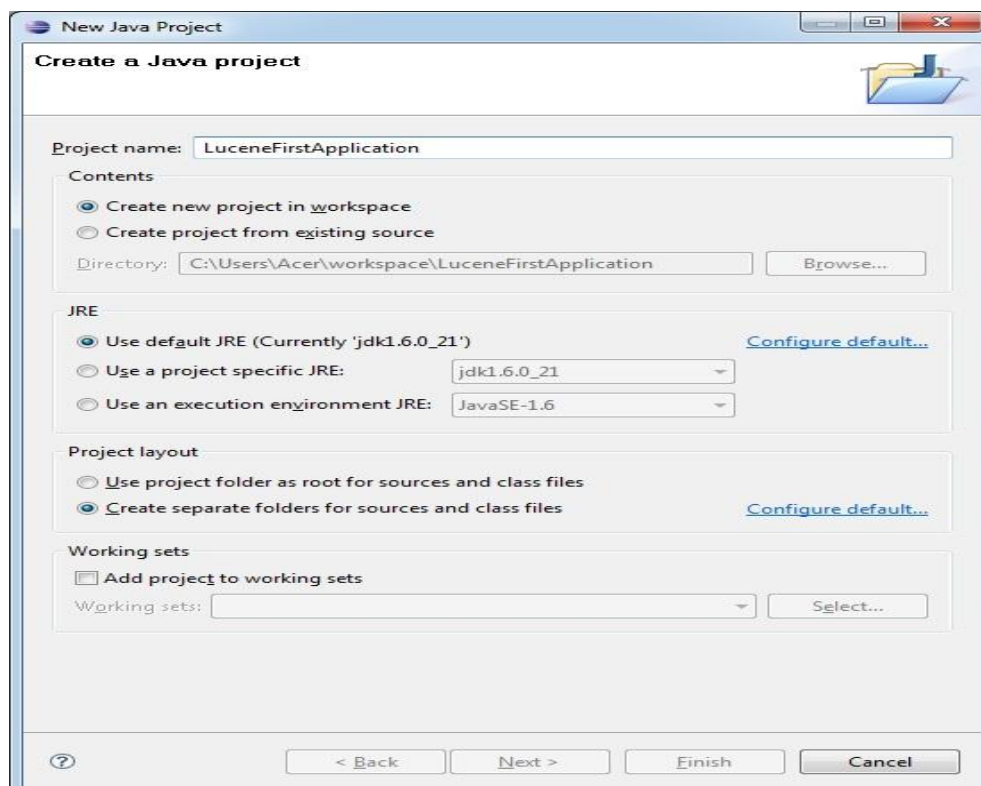
Lucene plays role in steps 2 to step 7 mentioned above and provides classes to do the required operations. In nutshell, lucene works as a heart of any search application and provides the vital operations pertaining to indexing and searching. Acquiring contents and displaying the results is left for the application part to handle. Let's start with first simple search application using lucene search library in next chapter.

Lucene – First Application

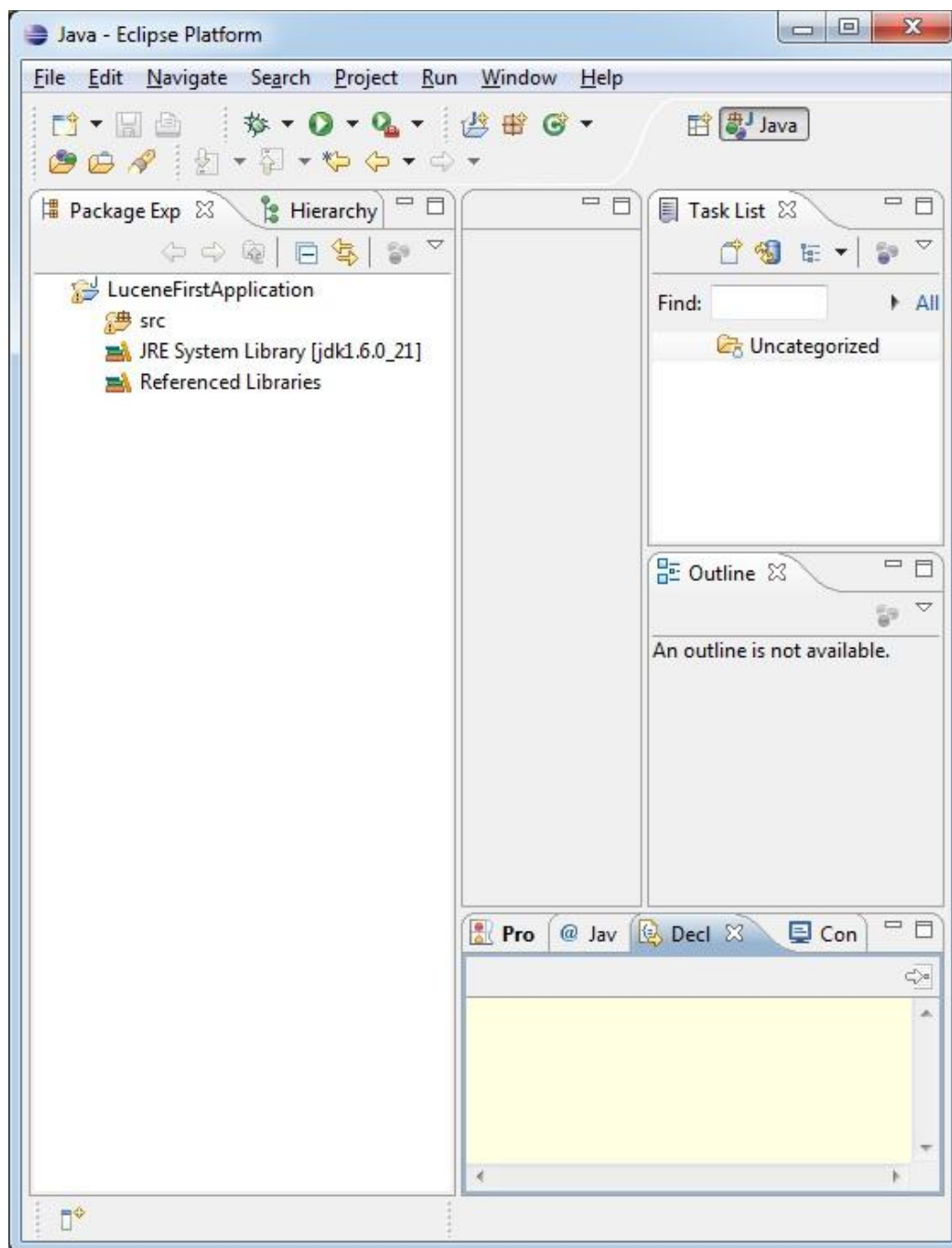
Let us start actual programming with Lucene Framework. So let us proceed to write a simple Search Application which will print number of search result found. We'll also see the list of indexes created during this process.

Step 1 - Create Java Project:

The first step is to create a simple Java Project using Eclipse IDE. Follow the option **File -> New -> Project** and finally select **Java Project** wizard from the wizard list. Now name your project as **LuceneFirstApplication** using the wizard window as follows:

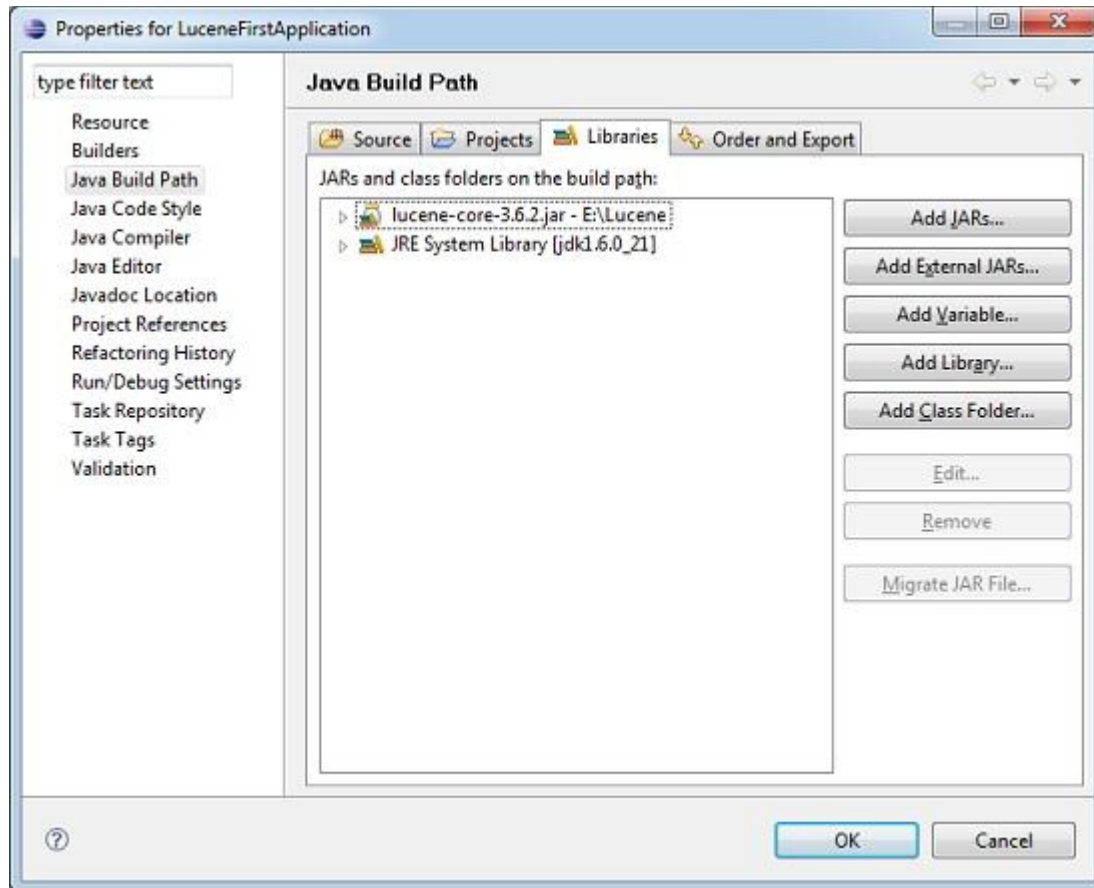


Once your project is created successfully, you will have following content in your **Project Explorer**:



Step 2 - Add Required Libraries:

As a second step let us add Lucene core Framework library in our project. To do this, right click on your project name **LuceneFirstApplication** and then follow the following option available in context menu: **Build Path -> Configure Build Path** to display the Java Build Path window as follows:



Now use **Add External JARs** button available under **Libraries** tab to add the following core JAR from Lucene installation directory:

- lucene-core-3.6.2

Step 3 - Create Source Files:

Now let us create actual source files under the **LuceneFirstApplication** project. First we need to create a package called **com.tutorialspoint.lucene**. To do this, right click on **src** in package explorer section and follow the option: **New -> Package**.

Next we will create **LuceneTester.java** and other java classes under the **com.tutorialspoint.lucene** package.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;
```

```

public Indexer(String indexDirectoryPath) throws IOException{
    //this directory will contain the indexes
    Directory indexDirectory =
        FSDirectory.open(new File(indexDirectoryPath));

    //create the indexer
    writer = new IndexWriter(indexDirectory,
        new StandardAnalyzer(Version.LUCENE_36), true,
        IndexWriter.MaxFieldLength.UNLIMITED);
}

public void close() throws CorruptIndexException, IOException{
    writer.close();
}

private Document getDocument(File file) throws IOException{
    Document document = new Document();

    //index file contents
    Field contentField = new Field(LuceneConstants.CONTENTES,
        new FileReader(file));
    //index file name
    Field fileNameField = new Field(LuceneConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);
    //index file path
    Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}

private void indexFile(File file) throws IOException{
    System.out.println("Indexing "+file.getCanonicalPath());
    Document document = getDocument(file);
    writer.addDocument(document);
}

public int createIndex(String dataDirPath, FileFilter filter)
    throws IOException{
    //get all files in the data directory
    File[] files = new File(dataDirPath).listFiles();

    for (File file : files) {
        if(!file.isDirectory()
            && !file.isHidden()
            && file.exists()
            && file.canRead()
            && filter.accept(file)
        ){
            indexFile(file);
        }
    }
    return writer.numDocs();
}
}

```

Searcher.java

This class is used to search the indexes created by Indexer to search the requested contents.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath)
        throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTES,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}
```

LuceneTester.java

This class is used to test the indexing and search capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
            tester.search("Mohan");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }

    private void search(String searchQuery) throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        TopDocs hits = searcher.search(searchQuery);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime));
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: "
                + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}
```

Step 4 - Data & Index directory creation











I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. **Test Data**. An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Step 5 - Running the Program:

Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

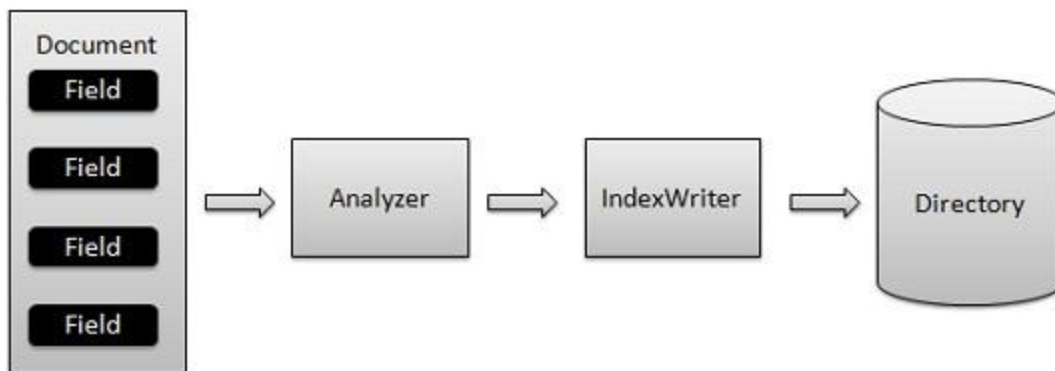
```
Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing E:\Lucene\Data\record8.txt
Indexing E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms
1 documents found. Time :0
File: E:\Lucene\Data\record4.txt
```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Indexing Classes

The indexing process is one of the core functionality provided by Lucene. Following diagram illustrates the indexing process and use of classes. IndexWriter is the most important and core component of the indexing process.



Indexing Process

We add *Document(s)* containing *Field(s)* to *IndexWriter* which analyzes the *Document(s)* using the *Analyzer* and then creates/open/edit indexes as required and store/update them in a *Directory*. *IndexWriter* is used to update or create indexes. It is not used to read indexes.

Indexing Classes:

Following is the list of commonly used classes during indexing process.

Sr. No.	Class & Description
1	IndexWriter This class acts as a core component which creates/updates indexes during indexing process.
2	Directory This class represents the storage location of the indexes.
3	Analyzer Analyzer class is responsible to analyze a document and get the tokens/words from the text which are to be indexed. Without analysis done, IndexWriter can not create index.
4	Document Document represents a virtual document with Fields where Field is object which can contain the physical document's contents, its meta data and so on. Analyzer can understand a Document only.
5	Field Field is the lowest unit or the starting point of the indexing process. It represents the key value pair relationship where a key is used to identify the value to be indexed. Say a field used to represent contents of a document will have key as "contents" and the value may contain the part or all of the text or numeric content of the document. Lucene can index only text or numeric contents only.

Lucene – IndexWriter

Introduction

This class acts as a core component which creates/updates indexes during indexing process.

Class declaration

Following is the declaration for **org.apache.lucene.index.IndexWriter** class:

```
public class IndexWriter
    extends Object
    implements Closeable, TwoPhaseCommit
```

Field

Following are the fields for **org.apache.lucene.index.IndexWriter** class:

- **static int DEFAULT_MAX_BUFFERED_DELETE_TERMS** -- Deprecated. Use IndexWriterConfig.DEFAULT_MAX_BUFFERED_DELETE_TERMS instead.
- **static int DEFAULT_MAX_BUFFERED_DOCS** -- Deprecated. Use IndexWriterConfig.DEFAULT_MAX_BUFFERED_DOCS instead.
- **static int DEFAULT_MAX_FIELD_LENGTH** -- Deprecated. See IndexWriterConfig.
- **static double DEFAULT_RAM_BUFFER_SIZE_MB** -- Deprecated. Use IndexWriterConfig.DEFAULT_RAM_BUFFER_SIZE_MB instead.
- **static int DEFAULT_TERM_INDEX_INTERVAL** -- Deprecated. Use IndexWriterConfig.DEFAULT_TERM_INDEX_INTERVAL instead.
- **static int DISABLE_AUTO_FLUSH** -- Deprecated. Use IndexWriterConfig.DISABLE_AUTO_FLUSH instead.
- **static int MAX_TERM_LENGTH** -- Absolute hard maximum length for a term.
- **static String WRITE_LOCK_NAME** -- Name of the write lock in the index.

- **static long WRITE_LOCK_TIMEOUT** -- Deprecated. Use `IndexWriterConfig.WRITE_LOCK_TIMEOUT` instead.

Class constructors

S.N.	Constructor & Description
1	IndexWriter(Directory d, Analyzer a, boolean create, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl) Deprecated. use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
2	IndexWriter(Directory d, Analyzer a, boolean create, IndexWriter.MaxFieldLength mfl) Deprecated. use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
3	IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl) Deprecated. use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
4	IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl, IndexCommit commit) Deprecated. use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
5	IndexWriter(Directory d, Analyzer a, IndexWriter.MaxFieldLength mfl) Deprecated. use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
6	IndexWriter(Directory d, IndexWriterConfig conf) Constructs a new <code>IndexWriter</code> per the settings given in <code>conf</code> .

Class methods

S.N.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	void addDocument(Document doc, Analyzer analyzer) Adds a document to this index, using the provided analyzer instead of the value of <code>getAnalyzer()</code> .
3	void addDocuments(Collection<Document> docs) Atomically adds a block of documents with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
4	void addDocuments(Collection<Document> docs, Analyzer analyzer) Atomically adds a block of documents, analyzed using the provided analyzer, with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
5	void addIndexes(Directory... dirs) Adds all segments from an array of indexes into this index.
6	void addIndexes(IndexReader... readers) Merges the provided indexes into this index.
7	void addIndexesNoOptimize(Directory... dirs) Deprecated. use <code>addIndexes(Directory...)</code> instead
8	void close() Commits all changes to an index and closes all associated files.
9	void close(boolean waitForMerges) Closes the index with or without waiting for currently running merges to finish.

10	void commit() Commits all pending changes (added & deleted documents, segment merges, added indexes, etc.) to the index, and syncs all referenced index files, such that a reader will see the changes and the index updates will survive an OS or machine crash or power loss.
11	void commit(Map<String,String> commitUserData) Commits all changes to the index, specifying a commitUserData Map (String -> String).
12	void deleteAll() Delete all documents in the index.
13	void deleteDocuments(Query... queries) Deletes the document(s) matching any of the provided queries.
14	void deleteDocuments(Query query) Deletes the document(s) matching the provided query.
15	void deleteDocuments(Term... terms) Deletes the document(s) containing any of the terms.
16	void deleteDocuments(Term term) Deletes the document(s) containing term.
17	void deleteUnusedFiles() Expert: remove any index files that are no longer used.
18	protected void doAfterFlush() A hook for extending classes to execute operations after pending added and deleted documents have been flushed to the Directory but before the change is committed (new segments_N file written).
19	protected void doBeforeFlush() A hook for extending classes to execute operations before pending added and deleted documents are flushed to the Directory.
20	protected void ensureOpen()
21	protected void ensureOpen(boolean includePendingClose) Used internally to throw an AlreadyClosedException if this IndexWriter has been closed.
22	void expungeDeletes() Deprecated.
23	void expungeDeletes(boolean doWait) Deprecated.
24	protected void flush(boolean triggerMerge, boolean applyAllDeletes) Flush all in-memory buffered updates (adds and deletes) to the Directory.
25	protected void flush(boolean triggerMerge, boolean flushDocStores, boolean flushDeletes) NOTE: flushDocStores is ignored now (hardwired to true); this method is only here for backwards compatibility
26	void forceMerge(int maxNumSegments) Forces merge policy to merge segments until there's <= maxNumSegments.
27	void forceMerge(int maxNumSegments, boolean doWait) Just like forceMerge(int), except you can specify whether the call should block until all merging completes.
28	void forceMergeDeletes() Forces merging of all segments that have deleted documents.
29	void forceMergeDeletes(boolean doWait) Just like forceMergeDeletes(), except you can specify whether the call should block until the operation completes.
30	Analyzer getAnalyzer()

	Returns the analyzer used by this index.
31	IndexWriterConfig getConfig() Returns the private IndexWriterConfig, cloned from the IndexWriterConfig passed to IndexWriter(Directory, IndexWriterConfig).
32	static PrintStream getDefaultInfoStream() Returns the current default infoStream for newly instantiated IndexWriters.
33	static long getDefaultWriteLockTimeout() Deprecated.use IndexWriterConfig.getDefaultWriteLockTimeout() instead
34	Directory getDirectory() Returns the Directory used by this index.
35	PrintStream getInfoStream() Returns the current infoStream in use by this writer.
36	int getMaxBufferedDeleteTerms() Deprecated.use IndexWriterConfig.getMaxBufferedDeleteTerms() instead
37	int getMaxBufferedDocs() Deprecated.use IndexWriterConfig.getMaxBufferedDocs() instead.
38	int getMaxFieldLength() Deprecated.use LimitTokenCountAnalyzer to limit number of tokens.
39	int getMaxMergeDocs() Deprecated.use LogMergePolicy.getMaxMergeDocs() directly.
40	IndexWriter.IndexReaderWarmer getMergedSegmentWarmer() Deprecated.use IndexWriterConfig.getMergedSegmentWarmer() instead.
41	int getMergeFactor() Deprecated.use LogMergePolicy.getMergeFactor() directly.
42	MergePolicy getMergePolicy() Deprecated.use IndexWriterConfig.getMergePolicy() instead
43	MergeScheduler getMergeScheduler() Deprecated.use IndexWriterConfig.getMergeScheduler() instead
44	Collection<SegmentInfo> getMergingSegments() Expert: to be used by a MergePolicy to a void selecting merges for segments already being merged.
45	MergePolicy.OneMerge getNextMerge() Expert: the MergeScheduler calls this method to retrieve the next merge requested by the MergePolicy
46	PayloadProcessorProvider getPayloadProcessorProvider() Returns the PayloadProcessorProvider that is used during segment merges to process payloads.
47	double getRAMBufferSizeMB() Deprecated.use IndexWriterConfig.getRAMBufferSizeMB() instead.
48	IndexReader getReader() Deprecated.Please use IndexReader.open(IndexWriter,boolean) instead.
49	IndexReader getReader(int termInfosIndexDivisor) Deprecated.Please use IndexReader.open(IndexWriter,boolean) instead. Furthermore, this method cannot guarantee the reader (and its sub-readers) will be opened with the termInfosIndexDivisor setting because some of them may have already been opened according to IndexWriterConfig.setReaderTermsIndexDivisor(int). You should set the requested termInfosIndexDivisor through IndexWriterConfig.setReaderTermsIndexDivisor(int) and use getReader().
50	int getReaderTermsIndexDivisor() Deprecated.use IndexWriterConfig.getReaderTermsIndexDivisor() instead.

51	Similarity getSimilarity() Deprecated.use IndexWriterConfig.getSimilarity() instead
52	int getTermIndexInterval() Deprecated.use IndexWriterConfig.getTermIndexInterval()
53	boolean getUseCompoundFile() Deprecated.use LogMergePolicy.getUseCompoundFile()
54	long getWriteLockTimeout() Deprecated.use IndexWriterConfig.getWriteLockTimeout()
55	boolean hasDeletions()
56	static boolean isLocked(Directory directory) Returns true iff the index in the named directory is currently locked.
57	int maxDoc() Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), not counting deletions.
58	void maybeMerge() Expert: asks the mergePolicy whether any merges are necessary now and if so, runs the requested merges and then iterate (test again if merges are needed) until no more merges are returned by the mergePolicy.
59	void merge(MergePolicy.OneMerge merge) Merges the indicated segments, replacing them in the stack with a single segment.
60	void message(String message) Prints a message to the infoStream (if non-null), prefixed with the identifying information for this writer and the thread that's calling it.
61	int numDeletedDocs(SegmentInfo info) Obtain the number of deleted docs for a pooled reader.
62	int numDocs() Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), and including deletions.
63	int numRamDocs() Expert: Return the number of documents currently buffered in RAM.
64	void optimize() Deprecated.
65	void optimize(boolean doWait) Deprecated.
66	void optimize(int maxNumSegments) Deprecated.
67	void prepareCommit() Expert: prepare for commit.
68	void prepareCommit(Map<String,String> commitUserData) Expert: prepare for commit, specifying commitUserData Map (String -> String).
69	long ramSizeInBytes() Expert: Return the total size of all index files currently cached in memory.
70	void rollback() Close the IndexWriter without committing any changes that have occurred since the last commit (or since it was opened, if commit hasn't been called).
71	String segString()

72	String segString(Iterable<SegmentInfo> infos)
73	String segString(SegmentInfo info)
74	static void setDefaultInfoStream(PrintStream infoStream) If non-null, this will be the default infoStream used by a newly instantiated IndexWriter.
75	static void setDefaultWriteLockTimeout(long writeLockTimeout) Deprecated.use IndexWriterConfig.setDefaultWriteLockTimeout(long) instead
76	void setInfoStream(PrintStream infoStream) If non-null, information about merges, deletes and a message when maxFieldLength is reached will be printed to this.
77	void setMaxBufferedDeleteTerms(int maxBufferedDeleteTerms) Deprecated.use IndexWriterConfig.setMaxBufferedDeleteTerms(int) instead.
78	void setMaxBufferedDocs(int maxBufferedDocs) Deprecated.use IndexWriterConfig.setMaxBufferedDocs(int) instead.
79	void setMaxFieldLength(int maxFieldLength) Deprecated.use LimitTokenCountAnalyzer instead. Note that the behavior slightly changed - the analyzer limits the number of tokens per token stream created, while this setting limits the total number of tokens to index. This only matters if you index many multi-valued fields though.
80	void setMaxMergeDocs(int maxMergeDocs) Deprecated.use LogMergePolicy.setMaxMergeDocs(int) directly.
81	void setMergedSegmentWarmer(IndexWriter.IndexReaderWarmer warmer) Deprecated.use IndexWriterConfig.setMergedSegmentWarmer(org.apache.lucene.index.IndexWriter.IndexReaderWarmer) instead.
82	void setMergeFactor(int mergeFactor) Deprecated.use LogMergePolicy.setMergeFactor(int) directly.
83	void setMergePolicy(MergePolicy mp) Deprecated.use IndexWriterConfig.setMergePolicy(MergePolicy) instead.
84	void setMergeScheduler(MergeScheduler mergeScheduler) Deprecated.use IndexWriterConfig.setMergeScheduler(MergeScheduler) instead
85	void setPayloadProcessorProvider(PayloadProcessorProvider pcp) Sets the PayloadProcessorProvider to use when merging payloads.
86	void setRAMBufferSizeMB(double mb) Deprecated.use IndexWriterConfig.setRAMBufferSizeMB(double) instead.
87	void setReaderTermsIndexDivisor(int divisor) Deprecated.use IndexWriterConfig.setReaderTermsIndexDivisor(int) instead.
88	void setSimilarity(Similarity similarity) Deprecated.use IndexWriterConfig.setSimilarity(Similarity) instead
89	void setTermIndexInterval(int interval) Deprecated.use IndexWriterConfig.setTermIndexInterval(int)
90	void setUseCompoundFile(boolean value) Deprecated.use LogMergePolicy.setUseCompoundFile(boolean).
91	void setWriteLockTimeout(long writeLockTimeout) Deprecated.use IndexWriterConfig.setWriteLockTimeout(long) instead
92	static void unlock(Directory directory) Forcibly unlocks the index in the named directory.
93	void updateDocument(Term term, Document doc)

	Updates a document by first deleting the document(s) containing term and then adding the new document.
94	void updateDocument(Term term, Document doc, Analyzer analyzer) Updates a document by first deleting the document(s) containing term and then adding the new document.
95	void updateDocuments(Term delTerm, Collection<Document> docs) Atomically deletes documents matching the provided delTerm and adds a block of documents with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
96	void updateDocuments(Term delTerm, Collection<Document> docs, Analyzer analyzer) Atomically deletes documents matching the provided delTerm and adds a block of documents, analyzed using the provided analyzer, with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
97	boolean verbose() Returns true if verbosing is enabled (i.e., infoStream !
98	void waitForMerges() Wait for any currently outstanding merges to finish.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

Lucene – Directory

Introduction

This class represents the storage location of the indexes and generally it is a list of files. These files are called index files. Index files are normally created once and then used for read operation or can be deleted.

Class declaration

Following is the declaration for **org.apache.lucene.store.Directory** class:

```
public abstract class Directory
    extends Object
    implements Closeable
```

Field

Following are the fields for **org.apache.lucene.store.Directory** class:

- **protected boolean isOpen**
- **protected LockFactory lockFactory** -- Holds the LockFactory instance (implements locking for this Directory instance).

Class constructors

S.N.	Constructor & Description
1	Directory()

Class methods

S.N.	Method & Description
1	void clearLock(String name) Attempt to clear (forcefully unlock and remove) the specified lock.
2	abstract void close() Closes the store.

	static void copy(Directory src, Directory dest, boolean closeDirSrc) Deprecated. Should be replaced with calls to copy(Directory, String, String) for every file that needs copying. You can use the following code:
3	<pre> IndexFileNameFilter filter = IndexFileNameFilter.getFilter(); for (String file : src.listAll()) { if (filter.accept(null, file)) { src.copy(dest, file, file); } } </pre>
4	void copy(Directory to, String src, String dest) Copies the file src to Directory to under the new file name dest.
5	abstract IndexOutput createOutput(String name) Creates a new, empty file in the directory with the given name.
6	abstract void deleteFile(String name) Removes an existing file in the directory.
7	protected void ensureOpen()
8	abstract boolean fileExists(String name) Returns true iff a file with the given name exists.
9	abstract long fileLength(String name) Returns the length of a file in the directory.
10	abstract long fileModified(String name) Deprecated.
11	LockFactory getLockFactory() Get the LockFactory that this Directory instance is using for its locking implementation.
12	String getLockID() Return a string identifier that uniquely differentiates this Directory instance from other Directory instances.
13	abstract String[] listAll() Returns an array of strings, one for each file in the directory.
14	Lock makeLock(String name) Construct a Lock.
15	abstract IndexInput openInput(String name) Returns a stream reading an existing file.
16	IndexInput openInput(String name, int bufferSize) Returns a stream reading an existing file, with the specified read buffer size.
17	void setLockFactory(LockFactory lockFactory) Set the LockFactory that this Directory instance should use for its locking implementation.
18	void sync(Collection<String> names) Ensure that any writes to these files are moved to stable storage.
19	void sync(String name) Deprecated. use sync(Collection) instead. For easy migration you can change your code to call sync(Collections.singleton(name))
20	String toString()
21	abstract void touchFile(String name) Deprecated. Lucene never uses this API; it will be removed in 4.0.

Methods inherited

This class inherits methods from the following classes:

- `java.lang.Object`

Lucene – Analyzer

Introduction

Analyzer class is responsible to analyze a document and get the tokens/words from the text which is to be indexed. Without analysis done, IndexWriter can not create index.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.Analyzer** class:

```
public abstract class Analyzer
    extends Object
    implements Closeable
```

Class constructors

S.N.	Constructor & Description
1	protected Analyzer()

Class methods

S.N.	Method & Description
1	void close() Frees persistent resources used by this Analyzer
2	int getOffsetGap(Fieldable field) Just like getPositionIncrementGap(java.lang.String), except for Token offsets instead.
3	int getPositionIncrementGap(String fieldName) Invoked before indexing a Fieldable instance if terms have already been added to that field.
4	protected Object getPreviousTokenStream() Used by Analyzers that implement reusableTokenStream to retrieve previously saved TokenStreams for re-use by the same thread.
5	TokenStream reusableTokenStream(String fieldName, Reader reader) Creates a TokenStream that is allowed to be re-used from the previous time that the same thread called this method.
6	protected void setPreviousTokenStream(Object obj) Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the

	same thread.
7	abstract TokenStream tokenStream(String fieldName, Reader reader) Creates a TokenStream which tokenizes all the text in the provided Reader.

Methods inherited

This class inherits methods from the following classes:

- `java.lang.Object`

Lucene – Document

Introduction

Document represents a virtual document with Fields where Field is object which can contain the physical document's contents, its meta-data and so on. Analyzer can understand a Document only.

Class declaration

Following is the declaration for **org.apache.lucene.document.Document** class:

```
public final class Document
    extends Object
    implements Serializable
```

Class constructors

S.N.	Constructor & Description
1	Document() Constructs a new document with no fields.

Class methods

S.N.	Method & Description
1	void clearLock(String name) Attempt to clear (forcefully unlock and remove) the specified lock.
2	void add(Fieldable field) Adds a field to a document.
3	String get(String name) Returns the string value of the field with the given name if any exist in this document, or null.
4	byte[] getBinaryValue(String name) Returns an array of bytes for the first (or only) field that has the name specified as the method parameter.
5	byte[][] getBinaryValues(String name) Returns an array of byte arrays for of the fields that have the name specified as the method parameter.
6	float getBoost() Returns, at indexing time, the boost factor as set by setBoost(float).
7	Field getField(String name) Deprecated. Use getFieldable(java.lang.String) instead and cast depending on data type.

8	Fieldable getFieldable(String name) Returns a field with the given name if any exist in this document, or null.
9	Fieldable[] getFieldables(String name) Returns an array of Fieldables with the given name.
10	List<Fieldable> getFields() Returns a List of all the fields in a document.
11	Field[] getFields(String name) Deprecated. Use getFieldable(java.lang.String) instead and cast depending on data type.
12	String[] getValues(String name) Returns an array of values of the field specified as the method parameter.
13	void removeField(String name) Removes field with the specified name from the document.
14	void removeFields(String name) Removes all fields with the given name from the document.
15	void setBoost(float boost) Sets a boost factor for hits on any field of this document.
16	String toString() Prints the fields of a document for human consumption.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

Lucene – Field

Introduction

Field is the lowest unit or the starting point of the indexing process. It represents the key value pair relationship where a key is used to identify the value to be indexed. Say a field used to represent contents of a document will have key as "contents" and the value may contain the part or all of the text or numeric content of the document.

Lucene can index only text or numeric contents only. This class represents the storage location of the indexes and generally it is a list of files. These files are called index files. Index files are normally created once and then used for read operation or can be deleted.

Class declaration

Following is the declaration for **org.apache.lucene.document.Field** class:

```
public final class Field
    extends AbstractField
    implements Fieldable, Serializable
```

Class constructors

S.N.	Constructor & Description
1	Field(String name, boolean internName, String value, Field.Store store, Field.Index index, Field.TermVector termVector) Create a field by specifying its name, value and how it will be saved in the index.
2	Field(String name, byte[] value) Create a stored field with binary value.
3	Field(String name, byte[] value, Field.Store store) Deprecated.
4	Field(String name, byte[] value, int offset, int length) Create a stored field with binary value.
5	Field(String name, byte[] value, int offset, int length, Field.Store store) Deprecated.
6	Field(String name, Reader reader) Create a tokenized and indexed field that is not stored.
7	Field(String name, Reader reader, Field.TermVector termVector) Create a tokenized and indexed field that is not stored, optionally with storing term vectors.
8	Field(String name, String value, Field.Store store, Field.Index index) Create a field by specifying its name, value and how it will be saved in the index.

9	Field(String name, String value, Field.Store store, Field.Index index, Field.TermVector termVector) Create a field by specifying its name, value and how it will be saved in the index.
10	Field(String name, TokenStream tokenStream) Create a tokenized and indexed field that is not stored.
11	Field(String name, TokenStream tokenStream, Field.TermVector termVector) Create a tokenized and indexed field that is not stored, optionally with storing term vectors.

Class methods

S.N.	Method & Description
1	void clearLock(String name) Attempt to clear (forcefully unlock and remove) the specified lock.
2	Reader readerValue() The value of the field as a Reader, or null.
3	void setTokenStream(TokenStream tokenStream) Expert: sets the token stream to be used for indexing and causes isIndexed() and isTokenized() to return true.
4	void setValue(byte[] value) Expert: change the value of this field.
5	void setValue(byte[] value, int offset, int length) Expert: change the value of this field.
6	void setValue(Reader value) Expert: change the value of this field.
7	void setValue(String value) Expert: change the value of this field.
8	String stringValue() The value of the field as a String, or null.
9	TokenStream tokenStreamValue() The TokenStream for this field to be used when indexing, or null.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.document.AbstractField
- java.lang.Object

Lucene – Searching Classes

Searching process is again one of the core functionality provided by Lucene. It's flow is similar to that of indexing process. Basic search of lucene can be made using following classes which can also be termed as foundation classes for all search related operations.

Searching Classes:

Following is the list of commonly used classes during searching process.

Sr. No.	Class & Description
1	IndexSearcher This class act as a core component which reads/searches indexes created after indexing process. It takes directory instance pointing to the location containing the indexes.
2	Term This class is the lowest unit of searching. It is similar to Field in indexing process.
3	Query Query is an abstract class and contains various utility methods and is the parent of all types of queries that lucene uses during search process.
4	TermQuery TermQuery is the most commonly used query object and is the foundation of many complex query that lucene can make use of.
5	TopDocs TopDocs points to the top N search results which matches the search criteria. It is simple container of pointers to point to documents which are output of search result.

Lucene – IndexSearcher

Introduction

This class acts as a core component which reads/searches indexes during searching process.

Class declaration

Following is the declaration for **org.apache.lucene.search.IndexSearcher** class:

```
public class IndexSearcher
    extends Searcher
```

Field

Following are the fields for **org.apache.lucene.index.IndexWriter** class:

- **protected int[] docStarts**
- **protected IndexReader[] subReaders**
- **protected IndexSearcher[] subSearchers**

Class constructors

S.N.	Constructor & Description
1	IndexSearcher(Directory path) Deprecated. Use IndexSearcher(IndexReader) instead.
2	IndexSearcher(Directory path, boolean readOnly) Deprecated. Use IndexSearcher(IndexReader) instead.
3	IndexSearcher(IndexReader r) Creates a searcher searching the provided index.
4	IndexSearcher(IndexReader r, ExecutorService executor) Runs searches for each segment separately, using the provided ExecutorService.

5	IndexSearcher(IndexReader reader, IndexReader[] subReaders, int[] docStarts) Expert: directly specify the reader, subReaders and their docID starts.
6	IndexSearcher(IndexReader reader, IndexReader[] subReaders, int[] docStarts, ExecutorService executor) Expert: directly specify the reader, subReaders and their docID starts, and an ExecutorService.

Class methods

S.N.	Method & Description
1	void close() Note that the underlying IndexReader is not closed, if IndexSearcher was constructed with IndexSearcher(IndexReader r).
2	Weight createNormalizedWeight(Query query) Creates a normalized weight for a top-level Query.
3	Document doc(int docID) Returns the stored fields of document i.
4	Document doc(int docID, FieldSelector fieldSelector) Get the Document at the nth position.
5	int docFreq(Term term) Returns total docFreq for this term.
6	Explanation explain(Query query, int doc) Returns an Explanation that describes how doc scored against query.
7	Explanation explain(Weight weight, int doc) Expert: low-level implementation method Returns an Explanation that describes how doc scored against weight.
8	protected void gatherSubReaders(List allSubReaders, IndexReader r)
9	IndexReader getIndexReader() Return the IndexReader this searches.
10	Similarity getSimilarity() Expert: Return the Similarity implementation used by this Searcher.
11	IndexReader[] getSubReaders() Returns the atomic subReaders used by this searcher.
12	int maxDoc() Expert: Returns one greater than the largest possible document number.
13	Query rewrite(Query original) Expert: called to re-write queries into primitive queries.
14	void search(Query query, Collector results) Lower-level search API.
15	void search(Query query, Filter filter, Collector results) Lower-level search API.
16	TopDocs search(Query query, Filter filter, int n) Finds the top n hits for query, applying filter if non-null.
17	TopFieldDocs search(Query query, Filter filter, int n, Sort sort) Search implementation with arbitrary sorting.

18	TopDocs search(Query query, int n) Finds the top n hits for query.
19	TopFieldDocs search(Query query, int n, Sort sort) Search implementation with arbitrary sorting and no filter.
20	void search(Weight weight, Filter filter, Collector collector) Lower-level search API.
21	TopDocs search(Weight weight, Filter filter, int nDocs) Expert: Low-level search implementation.
22	TopFieldDocs search(Weight weight, Filter filter, int nDocs, Sort sort) Expert: Low-level search implementation with arbitrary sorting.
23	protected TopFieldDocs search(Weight weight, Filter filter, int nDocs, Sort sort, boolean fillFields) Just like search(Weight, Filter, int, Sort), but you choose whether or not the fields in the returned FieldDoc instances should be set by specifying fillFields.
24	protected TopDocs search(Weight weight, Filter filter, ScoreDoc after, int nDocs) Expert: Low-level search implementation.
25	TopDocs searchAfter(ScoreDoc after, Query query, Filter filter, int n) Finds the top n hits for query, applying filter if non-null, where all results are after a previous result (after).
26	TopDocs searchAfter(ScoreDoc after, Query query, int n) Finds the top n hits for query where all results are after a previous result (after).
27	void setDefaultFieldSortScoring(boolean doTrackScores, boolean doMaxScore) By default, no scores are computed when sorting by field (using search(Query,Filter,int,Sort)).
28	void setSimilarity(Similarity similarity) Expert: Set the Similarity implementation used by this Searcher.
29	String toString()

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Searcher
- java.lang.Object

Lucene – Term

Introduction

This class is the lowest unit of searching. It is similar to Field in indexing process.

Class declaration

Following is the declaration for **org.apache.lucene.index.Term** class:

```
public final class Term
    extends Object
    implements Comparable, Serializable
```

Class constructors

S.N.	Constructor & Description
1	Term(String fld) Constructs a Term with the given field and empty text.
2	Term(String fld, String txt) Constructs a Term with the given field and text.

Class methods

S.N.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	int compareTo(Term other) Compares two terms, returning a negative integer if this term belongs before the argument, zero if this term is equal to the argument, and a positive integer if this term belongs after the argument.
3	Term createTerm(String text) Optimized construction of new Terms by reusing same field as this Term - avoids field.intern() overhead
4	boolean equals(Object obj)
5	String field() Returns the field of this term, an interned string.
6	int hashCode()

7	String text() Returns the text of this term.
8	String toString()

Methods inherited

This class inherits methods from the following classes:

- `java.lang.Object`

Lucene – Query

Introduction

Query is an abstract class and contains various utility methods and is the parent of all types of queries that lucene uses during search process.

Class declaration

Following is the declaration for **org.apache.lucene.search.Query** class:

```
public abstract class Query
    extends Object
    implements Serializable, Cloneable
```

Class constructors

S.N.	Constructor & Description
1	Query()

Class methods

S.N.	Method & Description
1	Object clone() Returns a clone of this query.
2	Query combine(Query[] queries) Expert: called when re-writing queries under MultiSearcher.
3	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
4	boolean equals(Object obj)
5	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
6	float getBoost() Gets the boost for this clause.
7	Similarity getSimilarity(Searcher searcher)

	Deprecated. Instead of using "runtime" subclassing/delegation, subclass the Weight instead.
8	int hashCode()
9	static Query mergeBooleanQueries(BooleanQuery... queries) Expert: merges the clauses of a set of BooleanQuery's into a single BooleanQuery.
10	Query rewrite(IndexReader reader) Expert: called to re-write queries into primitive queries.
11	void setBoost(float b) Sets the boost for this query clause to b.
12	String toString() Prints a query to a string.
13	abstract String toString(String field) Prints a query to a string, with field assumed to be the default field and omitted.
14	Weight weight(Searcher searcher) Deprecated. Never ever use this method in Weight implementations. Subclasses of Query should use <code>createWeight(org.apache.lucene.search.Searcher)</code> , instead.

Methods inherited

This class inherits methods from the following classes:

- `java.lang.Object`

Lucene – TermQuery

Introduction

TermQuery is the most commonly used query object and is the foundation of many complex queries that lucene can make use of.

Class declaration

Following is the declaration for **org.apache.lucene.search.TermQuery** class:

```
public class TermQuery
    extends Query
```

Class constructors

S.N.	Constructor & Description
1	TermQuery(Term t) Constructs a query for the term t.

Class methods

S.N.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
3	boolean equals(Object o) Returns true iff o is equal to this.
4	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
5	Term getTerm() Returns the term of this query.
6	int hashCode() Returns a hash code value for this object.
7	String toString(String field)

	Prints a user-readable version of this query.
--	---

Methods inherited

This class inherits methods from the following classes:

- `org.apache.lucene.search.Query`
- `java.lang.Object`

Lucene – TopDocs

Introduction

TopDocs points to the top N search results which matches the search criteria. It is simple container of pointers to point to documents which are output of search result.

Class declaration

Following is the declaration for **org.apache.lucene.search.TopDocs** class:

```
public class TopDocs
    extends Object
    implements Serializable
```

Field

Following are the fields for **org.apache.lucene.search.TopDocs** class:

- **ScoreDoc[] scoreDocs** -- The top hits for the query.
- **int totalHits** -- The total number of hits for the query.

Class constructors

S.N.	Constructor & Description
1	TopDocs(int totalHits, ScoreDoc[] scoreDocs, float maxScore)

Class methods

S.N.	Method & Description
1	getMaxScore() Returns the maximum score value encountered.
2	static TopDocs merge(Sort sort, int topN, TopDocs[] shardHits)

	Returns a new TopDocs, containing topN results across the provided TopDocs, sorting by the specified Sort.
3	void setMaxScore(float maxScore) Sets the maximum score value encountered.

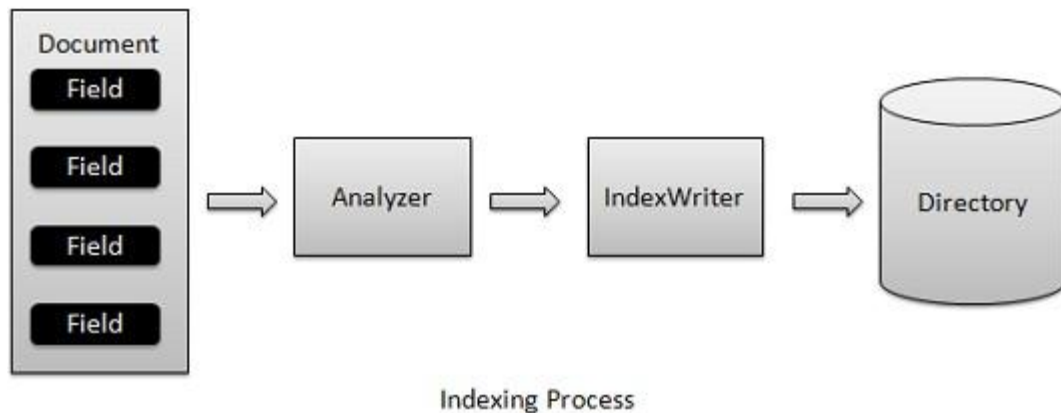
Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Query
- java.lang.Object

Lucene – Indexing Process

Indexing process is one of the core functionality provided by Lucene. Following diagram illustrates the indexing process and use of classes. IndexWriter is the most important and core component of the indexing process.



We add *Document(s)* containing *Field(s)* to *IndexWriter* which analyzes the *Document(s)* using the *Analyzer* and then creates/open/edit indexes as required and store/update them in a *Directory*. *IndexWriter* is used to update or create indexes. It is not used to read indexes.

Now we'll show you a step by step process to get a kick start in understanding of indexing process using a basic example.

Create a document

- Create a method to get a lucene document from a text file.
- Create various types of fields which are key value pairs containing keys as names and values as contents to be indexed.

- Set field to be analyzed or not. In our case, only contents is to be analyzed as it can contain data such as a, am, are, an etc. which are not required in search operations.
- Add the newly created fields to the document object and return it to the caller method.

```
private Document getDocument(File file) throws IOException{
    Document document = new Document();

    //index file contents
    Field contentField = new Field(LuceneConstants.CONTENTES,
        new FileReader(file));
    //index file name
    Field fileNameField = new Field(LuceneConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);
    //index file path
    Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}
```

Create a IndexWriter

- IndexWriter class acts as a core component which creates/updates indexes during indexing process.
- Create object of IndexWriter.
- Create a lucene directory which should point to location where indexes are to be stored.
- Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;

public Indexer(String indexDirectoryPath) throws IOException{
    //this directory will contain the indexes
    Directory indexDirectory =
        FSDirectory.open(new File(indexDirectoryPath));
    //create the indexer
    writer = new IndexWriter(indexDirectory,
        new StandardAnalyzer(Version.LUCENE_36),true,
        IndexWriter.MaxFieldLength.UNLIMITED);
}
```

Start Indexing process

```
private void indexFile(File file) throws IOException{
    System.out.println("Indexing "+file.getCanonicalPath());
}
```

```
Document document = getDocument(file);
writer.addDocument(document);
}
```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;
```

```

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36), true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private Document getDocument(File file) throws IOException{
        Document document = new Document();

        //index file contents
        Field contentField = new Field(LuceneConstants.CONTENTES,
            new FileReader(file));
        //index file name
        Field fileNameField = new Field(LuceneConstants.FILE_NAME,
            file.getName(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);
        //index file path
        Field filePathField = new Field(LuceneConstants.FILE_PATH,
            file.getCanonicalPath(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);

        document.add(contentField);
        document.add(fileNameField);
        document.add(filePathField);

        return document;
    }

    private void indexFile(File file) throws IOException{
        System.out.println("Indexing "+file.getCanonicalPath());
        Document document = getDocument(file);
        writer.addDocument(document);
    }
}

```

```

public int createIndex(String dataDirPath, FileFilter filter)
    throws IOException{
    //get all files in the data directory
    File[] files = new File(dataDirPath).listFiles();

    for (File file : files) {
        if(!file.isDirectory()
            && !file.isHidden()
            && file.exists()
            && file.canRead()
            && filter.accept(file)
        ){
            indexFile(file);
        }
    }
    return writer.numDocs();
}
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }
}

```


Data & Index directory creation











I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing E:\Lucene\Data\record8.txt
Indexing E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms
```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Indexing Operations

In this chapter, we'll discuss the four major operations of indexing. These operations are useful at various times and are used throughout of a software search application.

Indexing Operations:

Following is the list of commonly used operations during indexing process.

Sr. No.	Operation & Description
1	Add Document This operation is used in the initial stage of indexing process to create the indexes on the newly available contents.
2	Update Document This operation is used to update indexes to reflect the changes in the updated contents. It is similar to recreating the index.
3	Delete Document This operation is used to update indexes to exclude the documents which are not required to be indexed/searched.
4	Field Options Field options specifies a way or controls the way in which contents of a field are to be made searchable.

Lucene – Add Document

Add document is one of the core operations as part of indexing process. We add *Document(s)* containing *Field(s)* to *IndexWriter* where *IndexWriter* is used to update or create indexes.

Now we'll show you a step by step process to get a kick start in understanding of add document using a basic example.

Add a document to an index:

- Create a method to get a lucene document from a text file.
- Create various types of fields which are key-value pairs containing keys as names and values as contents to be indexed.
- Set field to be analyzed or not. In our case, only contents is to be analyzed as it can contain data such as a, am, are, an etc. which are not required in search operations.
- Add the newly created fields to the document object and return it to the caller method.

```
private Document getDocument(File file) throws IOException{
    Document document = new Document();

    //index file contents
    Field contentField = new Field(LuceneConstants.CONTENTES,
        new FileReader(file));
    //index file name
    Field fileNameField = new Field(LuceneConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);
    //index file path
    Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}
```

Create a IndexWriter

- IndexWriter class acts as a core component which creates/updates indexes during indexing process.
- Create object of IndexWriter.
- Create a lucene directory which should point to location where indexes are to be stored.
- Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;  
  
public Indexer(String indexDirectoryPath) throws IOException{  
    //this directory will contain the indexes  
    Directory indexDirectory =  
        FSDirectory.open(new File(indexDirectoryPath));  
    //create the indexer  
    writer = new IndexWriter(indexDirectory,  
        new StandardAnalyzer(Version.LUCENE_36), true,  
        IndexWriter.MaxFieldLength.UNLIMITED);  
}
```

Add document and start Indexing process

Following two are the ways to add the document.

- **addDocument(Document)** - Adds the document using the default analyzer (specified when index writer is created.)
- **addDocument(Document, Analyzer)** - Adds the document using the provided analyzer.

```
private void indexFile(File file) throws IOException{  
    System.out.println("Indexing "+file.getCanonicalPath());  
    Document document = getDocument(file);  
    writer.addDocument(document);  
}
```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
    }
}
```

```

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36), true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private Document getDocument(File file) throws IOException{
        Document document = new Document();

        //index file contents
        Field contentField = new Field(LuceneConstants.CONTENTES,
            new FileReader(file));
        //index file name
        Field fileNameField = new Field(LuceneConstants.FILE_NAME,
            file.getName(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);
        //index file path
        Field filePathField = new Field(LuceneConstants.FILE_PATH,
            file.getCanonicalPath(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);

        document.add(contentField);
        document.add(fileNameField);
        document.add(filePathField);

        return document;
    }

    private void indexFile(File file) throws IOException{
        System.out.println("Indexing "+file.getCanonicalPath());
        Document document = getDocument(file);
        writer.addDocument(document);
    }

    public int createIndex(String dataDirPath, FileFilter filter)
        throws IOException{
        //get all files in the data directory
        File[] files = new File(dataDirPath).listFiles();

        for (File file : files) {
            if(!file.isDirectory()
                && !file.isHidden()
                && file.exists()
                && file.canRead()
                && filter.accept(file)
            ){
                indexFile(file);
            }
        }
        return writer.numDocs();
    }
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```
package com.tutorialspoint.lucene;
```

```

import java.io.IOException;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }
}

```

Data & Index directory creation:

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:











Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing E:\Lucene\Data\record8.txt
Indexing E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms

```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Update Document

Update document is another important operation as part of indexing process. This operation is used when already indexed contents are updated and indexes become invalid. This operation is also known as re-indexing.

We update *Document(s)* containing *Field(s)* to *IndexWriter* where *IndexWriter* is used to update indexes.

Now we'll show you a step by step process to get a kick start in understanding of update document using a basic example.

Update a document to an index

- Create a method to update a lucene document from an updated text file.

```
private void updateDocument(File file) throws IOException{
    Document document = new Document();

    //update indexes for file
    writer.updateDocument(new Term(LuceneConstants.FILE_NAME, new FileReader(file)),document);
    writer.close();
}
```

Create a IndexWriter

- IndexWriter class acts as a core component which creates/updates indexes during indexing process.
- Create object of IndexWriter.
- Create a lucene directory which should point to location where indexes are to be stored.
- Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;

public Indexer(String indexDirectoryPath) throws IOException{
    //this directory will contain the indexes
    Directory indexDirectory =
        FSDirectory.open(new File(indexDirectoryPath));
    //create the indexer
}
```

```

writer = new IndexWriter(indexDirectory,
    new StandardAnalyzer(Version.LUCENE_36),true,
    IndexWriter.MaxFieldLength.UNLIMITED);
}

```

Update document and start reindexing process

Following two are the ways to update the document.

- **updateDocument(Term, Document)** - Delete the document containing the term and add the document using the default analyzer (specified when index writer is created.)
- **updateDocument(Term, Document,Analyzer)** - Delete the document containing the term and add the document using the provided analyzer.

```

private void indexFile(File file) throws IOException{
    System.out.println("Updating index for "+file.getCanonicalPath());
    updateDocument(file);
}

```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```

package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}

```

TextFileFilter.java

This class is used as a .txt file filter.

```

package com.tutorialspoint.lucene;

import java.io.File;

```

```
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36),true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private void updateDocument(File file) throws IOException{
        Document document = new Document();

        //update indexes for file
        writer.updateDocument(new Term(LuceneConstants.FILE_NAME, new FileReader(file)),document);
        writer.close();
    }

    private void indexFile(File file) throws IOException{
        System.out.println("Updating index: "+file.getCanonicalPath());
    }
}
```

```

        updateDocument(file);
    }

    public int createIndex(String dataDirPath, FileFilter filter)
        throws IOException{
        //get all files in the data directory
        File[] files = new File(dataDirPath).listFiles();

        for (File file : files) {
            if(!file.isDirectory()
                && !file.isHidden()
                && file.exists()
                && file.canRead()
                && filter.accept(file)
            ){
                indexFile(file);
            }
        }
        return writer.numDocs();
    }
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }
}

```

Data & Index directory creation











I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
Updating index for E:\Lucene\Data\record1.txt
Updating index for E:\Lucene\Data\record10.txt
Updating index for E:\Lucene\Data\record2.txt
Updating index for E:\Lucene\Data\record3.txt
Updating index for E:\Lucene\Data\record4.txt
Updating index for E:\Lucene\Data\record5.txt
Updating index for E:\Lucene\Data\record6.txt
Updating index for E:\Lucene\Data\record7.txt
Updating index for E:\Lucene\Data\record8.txt
Updating index for E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms
```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Delete Document

Delete document is another important operation as part of indexing process. This operation is used when already indexed contents are updated and indexes become invalid or indexes become very large in size then in order to reduce the size and update the index, delete operations are carried out.

We delete *Document(s)* containing *Field(s)* to *IndexWriter* where *IndexWriter* is used to update indexes.

Now we'll show you a step by step process to get a kick start in understanding of delete document using a basic example.

Delete a document from an index.

- Create a method to delete a lucene document of an obsolete text file.

```
private void deleteDocument(File file) throws IOException{  
    //delete indexes for a file  
    writer.deleteDocument(new Term(LuceneConstants.FILE_NAME,file.getName()));  
    writer.commit();  
}
```

Create a IndexWriter

- IndexWriter class acts as a core component which creates/updates indexes during indexing process.
- Create object of IndexWriter.
- Create a lucene directory which should point to location where indexes are to be stored.
- Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;  
public Indexer(String indexDirectoryPath) throws IOException{  
    //this directory will contain the indexes
```

```

Directory indexDirectory =
    FSDirectory.open(new File(indexDirectoryPath));
//create the indexer
writer = new IndexWriter(indexDirectory,
    new StandardAnalyzer(Version.LUCENE_36),true,
    IndexWriter.MaxFieldLength.UNLIMITED);
}

```

Delete document and start reindexing process

Following two are the ways to delete the document.

- **deleteDocuments(Term)** - Delete all the documents containing the term.
- **deleteDocuments(Term[])** - Delete all the documents containing any of the terms in the array.
- **deleteDocuments(Query)** - Delete all the documents matching the query.
- **deleteDocuments(Query[])** - Delete all the documents matching the query in the array.
- **deleteAll** - Delete all the documents.

```

private void indexFile(File file) throws IOException{
    System.out.println("Deleting index for "+file.getCanonicalPath());
    deleteDocument(file);
}

```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```

package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}

```


TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36),true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private void deleteDocument(File file) throws IOException{

```

```

//delete indexes for a file
writer.deleteDocuments(new Term(LuceneConstants.FILE_NAME,file.getName()));

        writer.commit();
    }

    private void indexFile(File file) throws IOException{
        System.out.println("Deleting index: "+file.getCanonicalPath());
        deleteDocument(file);
    }

    public int createIndex(String dataDirPath, FileFilter filter)
        throws IOException{
        //get all files in the data directory
        File[] files = new File(dataDirPath).listFiles();

        for (File file : files) {
            if(!file.isDirectory()
                && !file.isHidden()
                && file.exists()
                && file.canRead()
                && filter.accept(file)
            ){
                indexFile(file);
            }
        }
        return writer.numDocs();
    }
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
    }
}

```

```

    long endTime = System.currentTimeMillis();
    indexer.close();
}
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:











Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

Deleting index E:\Lucene\Data\record1.txt
Deleting index E:\Lucene\Data\record10.txt
Deleting index E:\Lucene\Data\record2.txt
Deleting index E:\Lucene\Data\record3.txt
Deleting index E:\Lucene\Data\record4.txt
Deleting index E:\Lucene\Data\record5.txt
Deleting index E:\Lucene\Data\record6.txt
Deleting index E:\Lucene\Data\record7.txt
Deleting index E:\Lucene\Data\record8.txt
Deleting index E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms

```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Field Options

Field is the most important and the foundation unit of indexing process. It is the actual object containing the contents to be indexed. When we add a field, lucene provides numerous controls on the field using Field Options which states how much a field is to be searchable.

We add *Document(s)* containing *Field(s)* to *IndexWriter* where *IndexWriter* is used to update or create indexes.

Now we'll show you a step by step process to get a kick start in understanding of various Field options using a basic example.

Various Field Options

- **Index.ANALYZED** - First analyze then do indexing. Used for normal text indexing. Analyzer will break the field's value into stream of tokens and each token is searchable separately.
- **Index.NOT_ANALYZED** - Don't analyze but do indexing. Used for complete text indexing for example person's names, URL etc.
- **Index.ANALYZED_NO_NORMS** - Variant of **Index.ANALYZED**. Analyzer will break the field's value into stream of tokens and each token is searchable separately but NORMs are not stored in the indexes. NORMs are used to boost searching but are sometime memory consuming.
- **Index.Index.NOT_ANALYZED_NO_NORMS** - Variant of **Index.NOT_ANALYZED**. Indexing is done but NORMs are not stored in the indexes.
- **Index.NO** - Field value is not searchable.

Use of Field Options

- Create a method to get a lucene document from a text file.
- Create various types of fields which are key value pairs containing keys as names and values as contents to be indexed.
- Set field to be analyzed or not. In our case, only contents is to be analyzed as it can contain data such as a, am, are, an etc. which are not required in search operations.
- Add the newly created fields to the document object and return it to the caller method.

```

private Document getDocument(File file) throws IOException{
    Document document = new Document();

    //index file contents
    Field contentField = new Field(LuceneConstants.CONTENTS,
        new FileReader(file));
    //index file name
    Field fileNameField = new Field(LuceneConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);
    //index file path
    Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}

```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```

package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}

```

TextFileFilter.java

This class is used as a .txt file filter.

```

package com.tutorialspoint.lucene;

```

```

import java.io.File;
import java.io.Filter;

public class TextFileFilter implements Filter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}

```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```

package com.tutorialspoint.lucene;

import java.io.File;
import java.io.Filter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36),true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private Document getDocument(File file) throws IOException{
        Document document = new Document();

        //index file contents
        Field contentField = new Field(LuceneConstants.CONTENTES,
            new FileReader(file));
        //index file name
        Field fileNameField = new Field(LuceneConstants.FILE_NAME,
            file.getName()),

```

```

        Field.Store.YES,Field.Index.NOT_ANALYZED);
//index file path
Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

document.add(contentField);
document.add(fileNameField);
document.add(filePathField);

return document;
}

private void indexFile(File file) throws IOException{
    System.out.println("Indexing "+file.getCanonicalPath());
    Document document = getDocument(file);
    writer.addDocument(document);
}

public int createIndex(String dataDirPath, FileFilter filter)
throws IOException{
    //get all files in the data directory
    File[] files = new File(dataDirPath).listFiles();

    for (File file : files) {
        if(!file.isDirectory()
            && !file.isHidden()
            && file.exists()
            && file.canRead()
            && filter.accept(file)
        ){
            indexFile(file);
        }
    }
    return writer.numDocs();
}
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}

private void createIndex() throws IOException{
    indexer = new Indexer(indexDir);
    int numIndexed;
    long startTime = System.currentTimeMillis();
    numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
    long endTime = System.currentTimeMillis();
    indexer.close();
    System.out.println(numIndexed+" File indexed, time taken: "
        +(endTime-startTime)+" ms");
}
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. Test Data. An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:











Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing E:\Lucene\Data\record8.txt
Indexing E:\Lucene\Data\record9.txt
10 File indexed, time taken: 109 ms

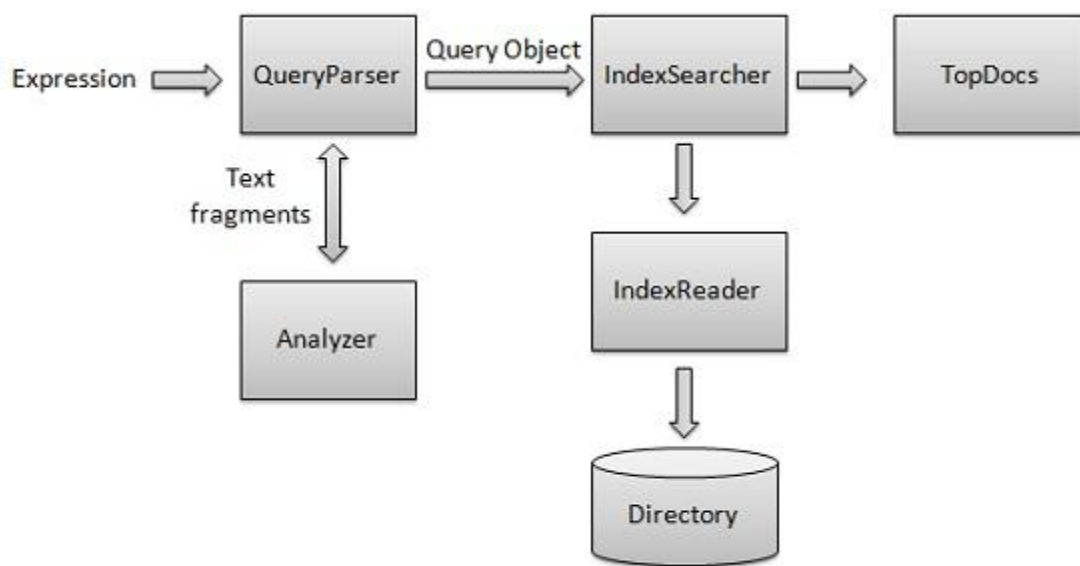
```

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB

Lucene – Search Operations

Searching process is one of the core functionality provided by Lucene. Following diagram illustrates the searching process and use of classes. `IndexSearcher` is the most important and core component of the searching process.



Searching Process

We first create *Directory(s)* containing *indexes* and then pass it to *IndexSearcher* which opens the *Directory* using *IndexReader*. Then we create a *Query* with a *Term* and make a search using *IndexSearcher* by passing the *Query* to the searcher. *IndexSearcher* returns a *TopDocs* object which contains the search details along with document ID(s) of the *Document* which is the result of the search operation.

Now we'll show you a step by step process to get a kick start in understanding of indexing process using a basic example.

Create a QueryParser

- QueryParser class parses the user entered input into lucene understandable format query.
- Create object of QueryParser.
- Initialize the QueryParser object created with a standard analyzer having version information and index name on which this query is to run.

```
QueryParser queryParser;  
  
public Searcher(String indexDirectoryPath) throws IOException{  
  
    queryParser = new QueryParser(Version.LUCENE_36,  
        LuceneConstants.CONTENTS,  
        new StandardAnalyzer(Version.LUCENE_36));  
}
```

Create a IndexSearcher

- IndexSearcher class acts as a core component which searcher indexes created during indexing process.
- Create object of IndexSearcher.
- Create a lucene directory which should point to location where indexes are to be stored.
- Initialize the IndexSearcher object created with the index directory

```
IndexSearcher indexSearcher;  
  
public Searcher(String indexDirectoryPath) throws IOException{  
    Directory indexDirectory =  
        FSDirectory.open(new File(indexDirectoryPath));  
    indexSearcher = new IndexSearcher(indexDirectory);  
}
```

Make search

- To start search, create a Query object by parsing search expression through QueryParser.
- Make search by calling IndexSearcher.search() method.

```
Query query;  
  
public TopDocs search( String searchQuery) throws IOException, ParseException{  
    query = queryParser.parse(searchQuery);  
    return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);  
}
```

Get the document

```
public Document getDocument(ScoreDoc scoreDoc)
```

```
throws CorruptIndexException, IOException{
return indexSearcher.doc(scoreDoc.doc);
}
```

Close IndexSearcher

```
public void close() throws IOException{
indexSearcher.close();
}
```

Example Application

Let us create a test Lucene application to test searching process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;
```

```

import org.apache.lucene.document.Document;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.search("Mohan");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void search(String searchQuery) throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        TopDocs hits = searcher.search(searchQuery);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + " ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\\Lucene\\Data**. **Test Data**. An index directory path should be created as **E:\\Lucene\\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

1 documents found. Time :29 ms
File: E:\\Lucene\\Data\\record4.txt

```

Lucene – Query Programming

As we've seen in previous chapter Lucene - Search Operation, Lucene uses IndexSearcher to make searches and it uses Query object created by QueryParser as input. In this chapter, we are going to discuss various types of Query objects and ways to create them programmatically. Creating different types of Query object gives control on the kind of search to be made.

Consider a case of Advanced Search, provided by many applications where users are given multiple options to confine the search results. By Query programming, we can achieve the same very easily.

Following is the list of Query types that we'll discuss in due course.

Sr. No.	Class & Description
1	TermQuery This class acts as a core component which creates/updates indexes during indexing process.
2	TermRangeQuery TermRangeQuery is the used when a range of textual terms are to be searched.
3	PrefixQuery PrefixQuery is used to match documents whose index starts with a specified string.
4	BooleanQuery BooleanQuery is used to search documents which are result of multiple queries using AND, OR or NOT operators.
5	PhraseQuery Phrase query is used to search documents which contain a particular sequence of terms.
6	WildcardQuery WildcardQuery is used to search documents using wildcards like '*' for any character sequence, '?' matching a single character.
7	FuzzyQuery FuzzyQuery is used to search documents using fuzzy implementation that is an approximate search based on edit distance algorithm.
8	MatchAllDocsQuery MatchAllDocsQuery as name suggests matches all the documents.

Lucene – TermQuery

Introduction

TermQuery is the most commonly used query object and is the foundation of many complex queries that Lucene can make use of. TermQuery is normally used to retrieve documents based on the key which is case sensitive.

Class declaration

Following is the declaration for **org.apache.lucene.search.TermQuery** class:

```
public class TermQuery
    extends Query
```

Class constructors

S.N.	Constructor & Description
1	TermQuery(Term t) Constructs a query for the term t.

Class methods

S.N.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
3	boolean equals(Object o) Returns true iff o is equal to this.
4	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
5	Term getTerm() Returns the term of this query.
6	int hashCode() Returns a hash code value for this object.
7	String toString(String field) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingTermQuery(String searchQuery) throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
    Query query = new TermQuery(term);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using TermQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
```



```
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TermQuery;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingTermQuery("record4.txt");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingTermQuery(String searchQuery) throws IOException, ParseException {
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
        //create the term query object
        Query query = new TermQuery(term);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}
```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\\Lucene\\Data**. [Test Data](#). An index directory path should be created as **E:\\Lucene\\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
1 documents found. Time :13 ms  
File: E:\Lucene\Data\record4.txt
```

Lucene – TermRangeQuery

Introduction

TermRangeQuery is the used when a range of textual terms are to be searched.

Class declaration

Following is the declaration for **org.apache.lucene.search.TermRangeQuery** class:

```
public class TermRangeQuery
    extends MultiTermQuery
```

Class constructors

S.N.	Constructor & Description
1	TermRangeQuery(String field, String lowerTerm, String upperTerm, boolean includeLower, boolean includeUpper) Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.
2	TermRangeQuery(String field, String lowerTerm, String upperTerm, boolean includeLower, boolean includeUpper, Collator collator) Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.

Class methods

S.N.	Method & Description
1	boolean equals(Object obj)
2	Collator getCollator() Returns the collator used to determine range inclusion, if any.
3	protected FilteredTermEnum getEnum(IndexReader reader) Construct the enumeration to be used, expanding the pattern term.
4	String getField() Returns the field name for this query.
5	String getLowerTerm() Returns the lower value of this range query.
6	String getUpperTerm() Returns the upper value of this range query.
7	int hashCode()
8	boolean includesLower()

	Returns true if the lower endpoint is inclusive.
9	boolean includesUpper() Returns true if the upper endpoint is inclusive.
10	String toString(String field) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.MultiTermQuery
- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingTermRangeQuery(String searchQueryMin,
    String searchQueryMax)throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create the term query object
    Query query = new TermRangeQuery(LuceneConstants.FILE_NAME,
        searchQueryMin,searchQueryMax,true,false);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using TermRangeQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{

```

```

        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TermRangeQuery;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingTermRangeQuery("record2.txt","record6.txt");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingTermRangeQuery(String searchQueryMin,
        String searchQueryMax)throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create the term query object
        Query query = new TermRangeQuery(LuceneConstants.FILE_NAME,
            searchQueryMin,searchQueryMax,true,false);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

```
}
```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
4 documents found. Time :17ms  
File: E:\Lucene\Data\record2.txt  
File: E:\Lucene\Data\record3.txt  
File: E:\Lucene\Data\record4.txt  
File: E:\Lucene\Data\record5.txt
```


Lucene – PrefixQuery

Introduction

PrefixQuery is used to match documents whose index starts with a specified string.

Class declaration

Following is the declaration for **org.apache.lucene.search.PrefixQuery** class:

```
public class PrefixQuery
    extends MultiTermQuery
```

Class constructors

S.N.	Constructor & Description
1	PrefixQuery(Term prefix) Constructs a query for the term starting with prefix.

Class methods

S.N.	Method & Description
1	boolean equals(Object o) Returns true iff o is equal to this.
2	protected FilteredTermEnum getEnum(IndexReader reader) Construct the enumeration to be used, expanding the pattern term
3	Term getPrefix() Returns the prefix of this query.
4	int hashCode() Returns a hash code value for this object.
5	String toString(String field) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.MultiTermQuery
- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingPrefixQuery(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
    Query query = new PrefixQuery(term);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using PrefixQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
```

```

import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;

```

```

import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.PrefixQuery;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingPrefixQuery("record1");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingPrefixQuery(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
        //create the term query object
        Query query = new PrefixQuery(term);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. **Test Data**. An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep **LuceneTester.java** file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

2 documents found. Time :20ms
File: E:\Lucene\Data\record1.txt
File: E:\Lucene\Data\record10.txt

Lucene – BooleanQuery

Introduction

BooleanQuery is used to search documents which are result of multiple queries using AND, OR or NOT operators.

Class declaration

Following is the declaration for **org.apache.lucene.search.BooleanQuery** class:

```
public class BooleanQuery
    extends Query
    implements Iterable<BooleanClause>
```

Fields

- **protected int minNrShouldMatch**

Class constructors

S.N.	Constructor & Description
1	BooleanQuery() Constructs an empty boolean query.
1	BooleanQuery(boolean disableCoord) Constructs an empty boolean query.

Class methods

S.N.	Method & Description
1	void add(BooleanClause clause) Adds a clause to a boolean query.
2	void add(Query query, BooleanClause.Occur occur) Adds a clause to a boolean query.
3	List<BooleanClause> clauses() Returns the list of clauses in this query.
4	Object clone() Returns a clone of this query.
5	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
6	boolean equals(Object o)

	Returns true iff o is equal to this.
7	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
8	BooleanClause[] getClauses() Returns the set of clauses in this query.
9	static int getMaxClauseCount() Return the maximum number of clauses permitted, 1024 by default.
10	int getMinimumNumberShouldMatch() Gets the minimum number of the optional BooleanClauses which must be satisfied.
11	int hashCode() Returns a hash code value for this object.
12	boolean isCoordDisabled() Returns true iff Similarity.coord(int,int) is disabled in scoring for this query instance.
13	Iterator<BooleanClause> iterator() Returns an iterator on the clauses in this query.
14	Query rewrite(IndexReader reader) Expert: called to re-write queries into primitive queries.
15	static void setMaxClauseCount(int maxClauseCount) Set the maximum number of clauses permitted per BooleanQuery.
16	void setMinimumNumberShouldMatch(int min) Specifies a minimum number of the optional BooleanClauses which must be satisfied.
17	String toString(String field) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingBooleanQuery(String searchQuery1,
    String searchQuery2)throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term1 = new Term(LuceneConstants.FILE_NAME, searchQuery1);
    //create the term query object
    Query query1 = new TermQuery(term1);

    Term term2 = new Term(LuceneConstants.FILE_NAME, searchQuery2);
    //create the term query object
    Query query2 = new PrefixQuery(term2);

    BooleanQuery query = new BooleanQuery();
    query.add(query1, BooleanClause.Occur.MUST_NOT);
    query.add(query2, BooleanClause.Occur.MUST);
}
```

```
//do the search
TopDocs hits = searcher.search(query);
long endTime = System.currentTimeMillis();

System.out.println(hits.totalHits +
    " documents found. Time : " + (endTime - startTime) + "ms");
for(ScoreDoc scoreDoc : hits.scoreDocs) {
    Document doc = searcher.getDocument(scoreDoc);
    System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
}
searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using BooleanQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
```



```

import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.BooleanClause;
import org.apache.lucene.search.PrefixQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TermQuery;
import org.apache.lucene.search.BooleanQuery;
import org.apache.lucene.search.TopDocs;

```

```

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingBooleanQuery("record1.txt", "record1");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingBooleanQuery(String searchQuery1,
        String searchQuery2) throws IOException, ParseException {
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term1 = new Term(LuceneConstants.FILE_NAME, searchQuery1);
        //create the term query object
        Query query1 = new TermQuery(term1);

        Term term2 = new Term(LuceneConstants.FILE_NAME, searchQuery2);
        //create the term query object
        Query query2 = new PrefixQuery(term2);

        BooleanQuery query = new BooleanQuery();
        query.add(query1, BooleanClause.Occur.MUST_NOT);
        query.add(query2, BooleanClause.Occur.MUST);

        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active

and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
1 documents found. Time :26ms  
File: E:\Lucene\Data\record10.txt
```

Lucene – PhraseQuery

Introduction

Phrase query is used to search documents which contain a particular sequence of terms.

Class declaration

Following is the declaration for **org.apache.lucene.search.PhraseQuery** class:

```
public class PhraseQuery
    extends Query
```

Class constructors

S.N.	Constructor & Description
1	PhraseQuery() Constructs an empty phrase query.

Class methods

S.N.	Method & Description
1	void add(Term term) Adds a term to the end of the query phrase.
2	void add(Term term, int position) Adds a term to the end of the query phrase.
3	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
4	boolean equals(Object o) Returns true iff o is equal to this.
5	void extractTerms(Set<Term> queryTerms) Expert: adds all terms occurring in this query to the terms set.
6	int[] getPositions() Returns the relative positions of terms in this phrase.
7	int getSlop() Returns the slop.
8	Term[] getTerms() Returns the set of terms in this phrase.
9	int hashCode()

	Returns a hash code value for this object.
10	Query rewrite(IndexReader reader) Expert: called to re-write queries into primitive queries.
11	void setSlop(int s) Sets the number of other words permitted between words in query phrase.
12	String toString(String f) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingPhraseQuery(String[] phrases)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();

    PhraseQuery query = new PhraseQuery();
    query.setSlop(0);

    for(String word:phrases){
        query.add(new Term(LuceneConstants.FILE_NAME,word));
    }

    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using PhraseQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep

	rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }
}
```

```

public TopDocs search(Query query) throws IOException, ParseException{
    return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
}

public Document getDocument(ScoreDoc scoreDoc)
    throws CorruptIndexException, IOException{
    return indexSearcher.doc(scoreDoc.doc);
}

public void close() throws IOException{
    indexSearcher.close();
}
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search PhraseQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            String[] phrases = new String[]{"record1.txt"};
            tester.searchUsingPhraseQuery(phrases);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingPhraseQuery(String[] phrases)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();

        PhraseQuery query = new PhraseQuery();
        query.setSlop(0);

        for(String word:phrases){
            query.add(new Term(LuceneConstants.FILE_NAME,word));
        }
    }
}

```

```

    }

    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. Test Data. An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

1 documents found. Time :14ms
File: E:\Lucene\Data\record1.txt

```


Lucene – WildcardQuery

Introduction

WildcardQuery is used to search documents using wildcards like '*' for any character sequence, '?' matching a single character.

Class declaration

Following is the declaration for **org.apache.lucene.search.WildcardQuery** class:

```
public class WildcardQuery
    extends MultiTermQuery
```

Fields

- protected Term term

Class constructors

S.N.	Constructor & Description
1	WildcardQuery(Term term)

Class methods

S.N.	Method & Description
1	boolean equals(Object obj)
2	protected FilteredTermEnum getEnum(IndexReader reader) Construct the enumeration to be used, expanding the pattern term.
3	Term getTerm() Returns the pattern term.
4	int hashCode()
5	String toString(String field) Prints a user-readable version of this query.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.MultiTermQuery
- org.apache.lucene.search.Query

- java.lang.Object

Usage

```
private void searchUsingWildCardQuery(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
    Query query = new WildcardQuery(term);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using WildcardQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;
```

```

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

```

```

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.WildcardQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingWildCardQuery("record1*");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingWildCardQuery(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
        //create the term query object
        Query query = new WildcardQuery(term);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. **Test Data**. An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run

your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
2 documents found. Time :47ms  
File: E:\Lucene\Data\record1.txt  
File: E:\Lucene\Data\record10.txt
```

Lucene – FuzzyQuery

Introduction

FuzzyQuery is used to search documents using fuzzy implementation that is an approximate search based on edit distance algorithm.

Class declaration

Following is the declaration for **org.apache.lucene.search.FuzzyQuery** class:

```
public class FuzzyQuery
    extends MultiTermQuery
```

Fields

- **static int defaultMaxExpansions**
- **static float defaultMinSimilarity**
- **static int defaultPrefixLength**
- **protected Term term**

Class constructors

S.N.	Constructor & Description
1	FuzzyQuery(Term term) Calls FuzzyQuery(term, 0.5f, 0, Integer.MAX_VALUE).
2	FuzzyQuery(Term term, float minimumSimilarity) Calls FuzzyQuery(term, minimumSimilarity, 0, Integer.MAX_VALUE).
3	FuzzyQuery(Term term, float minimumSimilarity, int prefixLength) Calls FuzzyQuery(term, minimumSimilarity, prefixLength, Integer.MAX_VALUE).
4	FuzzyQuery(Term term, float minimumSimilarity, int prefixLength, int maxExpansions) Create a new FuzzyQuery that will match terms with a similarity of at least minimum Similarity to term.

Class methods

1	boolean equals(Object obj)
2	protected FilteredTermEnum getEnum(IndexReader reader) Construct the enumeration to be used, expanding the pattern term.
3	float getMinSimilarity() Returns the minimum similarity that is required for this query to match.
4	int getPrefixLength()

	Returns the non-fuzzy prefix length.
5	Term getTerm() Returns the pattern term.
6	int hashCode()
7	String toString(String field) Prints a query to a string, with field assumed to be the default field and omitted.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.MultiTermQuery
- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingFuzzyQuery(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
    Query query = new FuzzyQuery(term);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.print("Score: " + scoreDoc.score + " ");
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using FuzzyQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.

4 Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTES,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }
}
```



```

    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.FuzzyQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingFuzzyQuery("cord3.txt");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingFuzzyQuery(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
        //create the term query object
        Query query = new FuzzyQuery(term);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {

```

```

        Document doc = searcher.getDocument(scoreDoc);
        System.out.print("Score: "+ scoreDoc.score + " ");
        System.out.println("File: "+ doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. Test Data. An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep **LuceneTester.java** file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

10 documents found. Time :78ms
Score: 1.3179655 File: E:\Lucene\Data\record3.txt
Score: 0.790779 File: E:\Lucene\Data\record1.txt
Score: 0.790779 File: E:\Lucene\Data\record2.txt
Score: 0.790779 File: E:\Lucene\Data\record4.txt
Score: 0.790779 File: E:\Lucene\Data\record5.txt
Score: 0.790779 File: E:\Lucene\Data\record6.txt
Score: 0.790779 File: E:\Lucene\Data\record7.txt
Score: 0.790779 File: E:\Lucene\Data\record8.txt
Score: 0.790779 File: E:\Lucene\Data\record9.txt
Score: 0.2635932 File: E:\Lucene\Data\record10.txt

```

Lucene – MatchAllDocsQuery

Introduction

MatchAllDocsQuery as name suggests matches all the documents.

Class declaration

Following is the declaration for **org.apache.lucene.search.MatchAllDocsQuery** class:

```
public class MatchAllDocsQuery  
    extends Query
```

Class constructors

S.N.	Constructor & Description
1	MatchAllDocsQuery()
2	MatchAllDocsQuery(String normsField)

Class methods

S.N.	Method & Description
1	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
2	boolean equals(Object o)
3	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
4	int hashCode()
5	String toString(String field) Prints a query to a string, with field assumed to be the default field and omitted.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.search.Query
- java.lang.Object

Usage

```
private void searchUsingMatchAllDocsQuery(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create the term query object
    Query query = new MatchAllDocsQuery(searchQuery);
    //do the search
    TopDocs hits = searcher.search(query);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.print("Score: " + scoreDoc.score + " ");
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Example Application

Let us create a test Lucene application to test search using MatchAllDocsQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;
```

```

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query) throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;

```

```

import org.apache.lucene.search.MatchAllDocsQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.searchUsingMatchAllDocsQuery("");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void searchUsingMatchAllDocsQuery(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create the term query object
        Query query = new MatchAllDocsQuery(searchQuery);
        //do the search
        TopDocs hits = searcher.search(query);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.print("Score: " + scoreDoc.score + " ");
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep **LuceneTester.java** file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

10 documents found. Time :9ms
Score: 1.0 File: E:\Lucene\Data\record1.txt
Score: 1.0 File: E:\Lucene\Data\record10.txt

```

Score: 1.0 File: E:\Lucene\Data\record2.txt
Score: 1.0 File: E:\Lucene\Data\record3.txt
Score: 1.0 File: E:\Lucene\Data\record4.txt
Score: 1.0 File: E:\Lucene\Data\record5.txt
Score: 1.0 File: E:\Lucene\Data\record6.txt
Score: 1.0 File: E:\Lucene\Data\record7.txt
Score: 1.0 File: E:\Lucene\Data\record8.txt
Score: 1.0 File: E:\Lucene\Data\record9.txt

Lucene – Analysis

As we've seen in one of the previous chapter *Lucene - Indexing Process*, Lucene uses *IndexWriter* which analyzes the *Document(s)* using the *Analyzer* and then creates/open/edit indexes as required. In this chapter, we are going to discuss various types of *Analyzer* objects and other relevant objects which are used during analysis process. Understanding Analysis process and how analyzers work will give you great insight over how lucene indexes the documents.

Following is the list of objects that we'll discuss in due course.

Sr. No.	Class & Description
1	Token Token represents text or word in a document with relevant details like its metadata (position, start offset, end offset, token type and its position increment).
2	TokenStream TokenStream is an output of analysis process and it comprises of series of tokens. It is an abstract class.
3	Analyzer This is abstract base class of for each and every type of Analyzer.
4	WhitespaceAnalyzer This analyzer splits the text in a document based on whitespace.
5	SimpleAnalyzer This analyzer splits the text in a document based on non-letter characters and then lowercase them.
6	StopAnalyzer This analyzer works similar to SimpleAnalyzer and remove the common words like 'a', 'an', 'the' etc.
7	StandardAnalyzer This is the most sophisticated analyzer and is capable of handling names, email address etc. It lowercases each token and removes common words and punctuation if any.

Lucene – Token

Introduction

Token represents text or word in a document with relevant details like its metadata(position, start offset, end offset, token type and its position increment).

Class declaration

Following is the declaration for **org.apache.lucene.analysis.Token** class:

```
public class Token
    extends TermAttributeImpl
    implements TypeAttribute, PositionIncrementAttribute,
        FlagsAttribute, OffsetAttribute,
        PayloadAttribute, PositionLengthAttribute
```

Fields

- **static AttributeSource.AttributeFactory TOKEN_ATTRIBUTE_FACTORY** - Convenience factory that returns Token as implementation for the basic attributes and return the default impl (with "Impl" appended) for all other attributes.

Class constructors

S.N.	Constructor & Description
1	Token() Constructs a Token with null text.
2	Token(char[] startTermBuffer, int termBufferOffset, int termBufferLength, int start, int end) Constructs a Token with the given term buffer (offset & length), start and end offsets
3	Token(int start, int end) Constructs a Token with null text and start & end offsets.
4	Token(int start, int end, int flags) Constructs a Token with null text and start & end offsets plus flags.
5	Token(int start, int end, String typ) Constructs a Token with null text and start & end offsets plus the Token type.
6	Token(String text, int start, int end) Constructs a Token with the given term text, and start & end offsets.
7	Token(String text, int start, int end, int flags) Constructs a Token with the given text, start and end offsets, & type.
8	Token(String text, int start, int end, String typ) Constructs a Token with the given text, start and end offsets, & type.

Class methods

S.N.	Method & Description
1	void clear() Resets the term text, payload, flags, and positionIncrement, startOffset, endOffset and token type to default.
2	Object clone() Shallow clone.
3	Token clone(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Makes a clone, but replaces the term buffer & start/end offset in the process.
4	void copyTo(AttributemImpl target) Copies the values from this Attribute into the passed-in target attribute.
5	int endOffset() Returns this Token's ending offset, one greater than the position of the last character corresponding to this token in the source text.
6	boolean equals(Object obj)
7	int getFlags() Get the bitset for any bits that have been set.
8	Payload getPayload() Returns this Token's payload.
9	int getPositionIncrement() Returns the position increment of this Token.
10	int getPositionLength() Get the position length.
11	int hashCode()
12	void reflectWith(AttributeReflector reflector) This method is for introspection of attributes, it should simply add the key/values this attribute holds to the given AttributeReflector.
13	Token reinit(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributemImpl.copyBuffer(char[], int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE
14	Token reinit(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributemImpl.copyBuffer(char[], int, int), setStartOffset(int), setEndOffset(int), setType(java.lang.String)
15	Token reinit(String newTerm, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributemImpl.append(CharSequence), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE
16	Token reinit(String newTerm, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributemImpl.append(CharSequence, int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE
17	Token reinit(String newTerm, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributemImpl.append(CharSequence, int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String)

18	Token reinit(String newTerm, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributImpl.append(CharSequence), setStartOffset(int), setEndOffset(int) setType(java.lang.String)
19	void reinit(Token prototype) Copy the prototype token's fields into this one.
20	void reinit(Token prototype, char[] newTermBuffer, int offset, int length) Copy the prototype token's fields into this one, with a different term.
21	void reinit(Token prototype, String newTerm) Copy the prototype token's fields into this one, with a different term.
22	void setEndOffset(int offset) Set the ending offset.
23	void setFlags(int flags)
24	void setOffset(int startOffset, int endOffset) Set the starting and ending offset.
25	void setPayload(Payload payload) Sets this Token's payload.
26	void setPositionIncrement(int positionIncrement) Set the position increment.
27	void setPositionLength(int positionLength) Set the position length.
28	void setStartOffset(int offset) Set the starting offset.
29	void setType(String type) Set the lexical type.
30	int startOffset() Returns this Token's starting offset, the position of the first character corresponding to this token in the source text.
31	String type() Returns this Token's lexical type.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.tokenattributes.TermAttributImpl
- org.apache.lucene.analysis.tokenattributes.CharTermAttributImpl
- org.apache.lucene.util.AttributImpl
- java.lang.Object

Lucene – TokenStream

Introduction

TokenStream is an output of analysis process and it comprises of series of tokens. It is an abstract class.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.TokenStream** class:

```
public abstract class TokenStream
    extends AttributeSource
    implements Closeable
```

Class constructors

S.N.	Constructor & Description
1	protected TokenStream() A TokenStream using the default attribute factory.
2	protected TokenStream(AttributeSource.AttributeFactory factory) A TokenStream using the supplied AttributeFactory for creating new Attribute instances.
3	protected TokenStream(AttributeSource input) A TokenStream that uses the same attributes as the supplied one.

Class methods

S.N.	Method & Description
1	void close() Releases resources associated with this stream.
2	void end() This method is called by the consumer after the last token has been consumed, after incrementToken() returned false (using the new TokenStream API).
3	abstract boolean incrementToken() Consumers (i.e., IndexWriter) use this method to advance the stream to the next token.
4	void reset() Resets this stream to the beginning.

Methods inherited

This class inherits methods from the following classes:

- `org.apache.lucene.util.AttributeSource`
- `java.lang.Object`

Lucene – Analyzer

Introduction

Analyzer class is responsible to analyze a document and get the tokens/words from the text which is to be indexed. Without analysis done, IndexWriter can not create index.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.Analyzer** class:

```
public abstract class Analyzer
    extends Object
    implements Closeable
```

Class constructors

S.N.	Constructor & Description
1	protected Analyzer()

Class methods

S.N.	Method & Description
1	void close() Frees persistent resources used by this Analyzer
2	int getOffsetGap(Fieldable field) Just like getPositionIncrementGap(java.lang.String), except for Token offsets instead.
3	int getPositionIncrementGap(String fieldName) Invoked before indexing a Fieldable instance if terms have already been added to that field.
4	protected Object getPreviousTokenStream() Used by Analyzers that implement reusableTokenStream to retrieve previously saved TokenStreams for re-use by the same thread.
5	TokenStream reusableTokenStream(String fieldName, Reader reader) Creates a TokenStream that is allowed to be re-used from the previous time that the same thread called this method.
6	protected void setPreviousTokenStream(Object obj) Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the same thread.
7	abstract TokenStream tokenStream(String fieldName, Reader reader) Creates a TokenStream which tokenizes all the text in the provided Reader.

Methods inherited

This class inherits methods from the following classes:

- `java.lang.Object`

Lucene – WhitespaceAnalyzer

Introduction

This is the most sophisticated analyzer and is capable of handling names, email address etc. It lowercases each token and removes common words and punctuation if any.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.StandardAnalyzer** class:

```
public final class StandardAnalyzer
    extends StopwordAnalyzerBase
```

Fields

- **static int DEFAULT_MAX_TOKEN_LENGTH** - Default maximum allowed token length
- **static Set<?> STOP_WORDS_SET** - An unmodifiable set containing some common English words that are usually not useful for searching.

Class constructors

S.N.	Constructor & Description
1	StandardAnalyzer(Version matchVersion) Builds an analyzer with the default stop words (STOP_WORDS_SET).
1	StandardAnalyzer(Version matchVersion, File stopwords) Deprecated. Use StandardAnalyzer(Version, Reader) instead.
1	StandardAnalyzer(Version matchVersion, Reader stopwords) Builds an analyzer with the stop words from the given reader.
1	StandardAnalyzer(Version matchVersion, Set<?> stopWords) Builds an analyzer with the given stop words.

Class methods

S.N.	Method & Description
1	protected ReusableAnalyzerBase.TokenStreamComponents createComponents(String fieldName, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.
2	int getMaxTokenLength()
3	void setMaxTokenLength(int length) Set maximum allowed token length.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.StopwordAnalyzerBase
- org.apache.lucene.analysis.ReusableAnalyzerBase
- org.apache.lucene.analysis.Analyzer
- java.lang.Object

Usage

```
private void displayTokenUsingStandardAnalyzer() throws IOException{
    String text
        = "Lucene is simple yet powerful java based search library.";
    Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_36);
    TokenStream tokenStream
        = analyzer.tokenStream(LuceneConstants.CONTENTES,
            new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ";
    }
}
```

Example Application

Let us create a test Lucene application to test search using BooleanQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;
import java.io.StringReader;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.StandardAnalyzer;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;
import org.apache.lucene.util.Version;

public class LuceneTester {

    public static void main(String[] args) {
        LuceneTester tester;

        tester = new LuceneTester();

        try {
            tester.displayTokenUsingStandardAnalyzer();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void displayTokenUsingStandardAnalyzer() throws IOException{
        String text
            = "Lucene is simple yet powerful java based search library.";
        Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_36);
        TokenStream tokenStream = analyzer.tokenStream(
            LuceneConstants.CONTENT, new StringReader(text));
        TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
        while(tokenStream.incrementToken()) {
            System.out.print "[" + term.term() + " ] ";
        }
    }
}
```

Running the Program:

Once you are done with creating source, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
[lucene] [simple] [yet] [powerful] [java] [based] [search] [library]
```

Lucene – SimpleAnalyzer

Introduction

This analyzer splits the text in a document based on non-letter characters and then lowercase them.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.SimpleAnalyzer** class:

```
public final class SimpleAnalyzer
    extends ReusableAnalyzerBase
```

Class constructors

S.N.	Constructor & Description
1	SimpleAnalyzer() Deprecated. use SimpleAnalyzer(Version) instead
2	SimpleAnalyzer(Version matchVersion) Creates a new SimpleAnalyzer

Class methods

S.N.	Method & Description
1	protected ReusableAnalyzerBase.TokenStreamComponents createComponents(String fieldName, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.ReusableAnalyzerBase
- org.apache.lucene.analysis.Analyzer
- java.lang.Object

Usage

```
private void displayTokenUsingSimpleAnalyzer() throws IOException{
    String text = "Lucene is simple yet powerful java based search library.";
    Analyzer analyzer = new SimpleAnalyzer(Version.LUCENE_36);
    TokenStream tokenStream = analyzer.tokenStream(
        LuceneConstants.CONTENTS,
        new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ";
    }
}
```

Example Application

Let us create a test Lucene application to test search using BooleanQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;
import java.io.StringReader;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;
import org.apache.lucene.util.Version;

public class LuceneTester {
```

```

public static void main(String[] args) {
    LuceneTester tester;

    tester = new LuceneTester();

    try {
        tester.displayTokenUsingSimpleAnalyzer();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private void displayTokenUsingSimpleAnalyzer() throws IOException{
    String text =
        "Lucene is simple yet powerful java based search library.";
    Analyzer analyzer = new SimpleAnalyzer(Version.LUCENE_36);
    TokenStream tokenStream = analyzer.tokenStream(
        LuceneConstants.CONTENTES, new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ");
    }
}
}

```

Running the Program:

Once you are done with creating source, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
[lucene] [is] [simple] [yet] [powerful] [java] [based] [search] [library]
```

Lucene – StopAnalyzer

Introduction

This analyzer works similar to SimpleAnalyzer and remove the common words like 'a','an','the' etc.

Class declaration

Following is the declaration for **org.apache.lucene.analysis.StopAnalyzer** class:

```
public final class StopAnalyzer
    extends StopwordAnalyzerBase
```

Fields

- **static Set<?> ENGLISH_STOP_WORDS_SET** - An unmodifiable set containing some common English words that are not usually useful for searching.

Class constructors

S.N.	Constructor & Description
1	StopAnalyzer(Version matchVersion) Builds an analyzer which removes words in ENGLISH_STOP_WORDS_SET.
2	StopAnalyzer(Version matchVersion, File stopwordsFile) Builds an analyzer with the stop words from the given file.
3	StopAnalyzer(Version matchVersion, Reader stopwords) Builds an analyzer with the stop words from the given reader.
4	StopAnalyzer(Version matchVersion, Set<?> stopWords) Builds an analyzer with the stop words from the given set.

Class methods

S.N.	Method & Description
1	protected ReusableAnalyzerBase.TokenStreamComponents createComponents(String fieldName, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents used to tokenize all the text in the provided Reader.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.StopwordAnalyzerBase

- org.apache.lucene.analysis.ReusableAnalyzerBase
- org.apache.lucene.analysis.Analyzer
- java.lang.Object

Usage

```
private void displayTokenUsingStopAnalyzer() throws IOException{
    String text
        = "Lucene is simple yet powerful java based search library.";
    Analyzer analyzer = new StopAnalyzer(Version.LUCENE_36);
    TokenStream tokenStream
        = analyzer.tokenStream(LuceneConstants.CONTENTS,
            new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ";
    }
}
```

Example Application

Let us create a test Lucene application to test search using BooleanQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;
import java.io.StringReader;
```

```

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;
import org.apache.lucene.util.Version;

public class LuceneTester {

    public static void main(String[] args) {
        LuceneTester tester;

        tester = new LuceneTester();

        try {
            tester.displayTokenUsingStopAnalyzer();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void displayTokenUsingStopAnalyzer() throws IOException{
        String text
            = "Lucene is simple yet powerful java based search library.";
        Analyzer analyzer = new StopAnalyzer(Version.LUCENE_36);
        TokenStream tokenStream = analyzer.tokenStream(
            LuceneConstants.CONTENTES, new StringReader(text));
        TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
        while(tokenStream.incrementToken()) {
            System.out.print("[ " + term.term() + " ] ");
        }
    }
}

```

Running the Program:

Once you are done with creating source, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
[lucene] [simple] [yet] [powerful] [java] [based] [search] [library]
```


Lucene – StandardAnalyzer

Introduction

This is the most sophisticated analyzer and is capable of handling names, email address etc. It lowercases each token and removes common words and punctuation if any.

Class declaration

Following is the declaration for org.apache.lucene.analysis.StandardAnalyzer class:

```
public final class StandardAnalyzer
    extends StopwordAnalyzerBase
```

Fields

- **static int DEFAULT_MAX_TOKEN_LENGTH** - Default maximum allowed token length
- **static Set<?> STOP_WORDS_SET** - An unmodifiable set containing some common English words that are usually not useful for searching.

Class constructors

S.N.	Constructor & Description
1	StandardAnalyzer(Version matchVersion) Builds an analyzer with the default stop words (STOP_WORDS_SET).
2	StandardAnalyzer(Version matchVersion, File stopwords) Deprecated. Use StandardAnalyzer(Version, Reader) instead.
3	StandardAnalyzer(Version matchVersion, Reader stopwords) Builds an analyzer with the stop words from the given reader.
4	StandardAnalyzer(Version matchVersion, Set<?> stopWords) Builds an analyzer with the given stop words.

Class methods

S.N.	Method & Description
1	protected ReusableAnalyzerBase.TokenStreamComponents createComponents(String fieldName, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.
2	int getMaxTokenLength()
3	void setMaxTokenLength(int length) Set maximum allowed token length.

Methods inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.StopwordAnalyzerBase
- org.apache.lucene.analysis.ReusableAnalyzerBase
- org.apache.lucene.analysis.Analyzer
- java.lang.Object

Usage

```
private void displayTokenUsingStandardAnalyzer() throws IOException{
    String text
        = "Lucene is simple yet powerful java based search library.";
    Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_36);
    TokenStream tokenStream
        = analyzer.tokenStream(LuceneConstants.CONTENTS,
            new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ";
    }
}
```

Example Application

Let us create a test Lucene application to test search using BooleanQuery.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```
package com.tutorialspoint.lucene;

import java.io.IOException;
import java.io.StringReader;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.StandardAnalyzer;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;
import org.apache.lucene.util.Version;

public class LuceneTester {

    public static void main(String[] args) {
        LuceneTester tester;

        tester = new LuceneTester();

        try {
            tester.displayTokenUsingStandardAnalyzer();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void displayTokenUsingStandardAnalyzer() throws IOException{
        String text
            = "Lucene is simple yet powerful java based search library.";
        Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_36);
        TokenStream tokenStream = analyzer.tokenStream(
            LuceneConstants.CONTENT, new StringReader(text));
        TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
        while(tokenStream.incrementToken()) {
            System.out.print "[" + term.term() + " ] ";
        }
    }
}
```

Running the Program:

Once you are done with creating source, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
[lucene] [simple] [yet] [powerful] [java] [based] [search] [library]
```

Lucene – Sorting

Introduction

In this chapter we will look into the sorting orders in which lucene gives the search results by default or can be manipulated as required.

Sorting By Relevance

This is default sorting mode used by lucene. Lucene provides results by the most relevant hit at the top.

```
private void sortUsingRelevance(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
    Query query = new FuzzyQuery(term);
    searcher.setDefaultFieldSortScoring(true, false);
    //do the search
    TopDocs hits = searcher.search(query, Sort.RELEVANCE);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits +
        " documents found. Time : " + (endTime - startTime) + "ms");
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.print("Score: " + scoreDoc.score + " ");
        System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}
```

Sorting By IndexOrder

This is sorting mode used by lucene in which first document indexed is shown first in the search results.

```
private void sortUsingIndex(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
    Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
    //create the term query object
```

```

Query query = new FuzzyQuery(term);
searcher.setDefaultFieldSortScoring(true, false);
//do the search
TopDocs hits = searcher.search(query, Sort.INDEXORDER);
long endTime = System.currentTimeMillis();

System.out.println(hits.totalHits +
    " documents found. Time : " + (endTime - startTime) + "ms");
for(ScoreDoc scoreDoc : hits.scoreDocs) {
    Document doc = searcher.getDocument(scoreDoc);
    System.out.print("Score: " + scoreDoc.score + " ");
    System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
}
searcher.close();
}

```

Example Application

Let us create a test Lucene application to test sorting process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First Application</i> chapter as such for this chapter to understand searching process.
2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```

package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}

```

Searcher.java

This class is used to read the indexes made on raw data and searches data using lucene library.

```

package com.tutorialspoint.lucene;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;

```

```

import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.Sort;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException{
        Directory indexDirectory
            = FSDirectory.open(new File(indexDirectoryPath));
        indexSearcher = new IndexSearcher(indexDirectory);
        queryParser = new QueryParser(Version.LUCENE_36,
            LuceneConstants.CONTENTS,
            new StandardAnalyzer(Version.LUCENE_36));
    }

    public TopDocs search( String searchQuery)
        throws IOException, ParseException{
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query)
        throws IOException, ParseException{
        return indexSearcher.search(query, LuceneConstants.MAX_SEARCH);
    }

    public TopDocs search(Query query,Sort sort)
        throws IOException, ParseException{
        return indexSearcher.search(query,
            LuceneConstants.MAX_SEARCH,sort);
    }

    public void setDefaultFieldSortScoring(boolean doTrackScores,
        boolean doMaxScores){
        indexSearcher.setDefaultFieldSortScoring(
            doTrackScores,doMaxScores);
    }

    public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException{
        return indexSearcher.doc(scoreDoc.doc);
    }

    public void close() throws IOException{
        indexSearcher.close();
    }
}

```

LuceneTester.java

This class is used to test the searching capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.FuzzyQuery;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.Sort;
import org.apache.lucene.search.TopDocs;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;
    Searcher searcher;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.sortUsingRelevance("cord3.txt");
            tester.sortUsingIndex("cord3.txt");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private void sortUsingRelevance(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        //create a term to search file name
        Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
        //create the term query object
        Query query = new FuzzyQuery(term);
        searcher.setDefaultFieldSortScoring(true, false);
        //do the search
        TopDocs hits = searcher.search(query, Sort.RELEVANCE);
        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits +
            " documents found. Time : " + (endTime - startTime) + "ms");
        for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
            System.out.print("Score: " + scoreDoc.score + " ");
            System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
        }
        searcher.close();
    }

    private void sortUsingIndex(String searchQuery)
        throws IOException, ParseException{
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
    }

```

```

//create a term to search file name
Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
//create the term query object
Query query = new FuzzyQuery(term);
searcher.setDefaultFieldSortScoring(true, false);
//do the search
TopDocs hits = searcher.search(query, Sort.INDEXORDER);
long endTime = System.currentTimeMillis();

System.out.println(hits.totalHits +
" documents found. Time : " + (endTime - startTime) + "ms");
for(ScoreDoc scoreDoc : hits.scoreDocs) {
    Document doc = searcher.getDocument(scoreDoc);
    System.out.print("Score: " + scoreDoc.score + " ");
    System.out.println("File: " + doc.get(LuceneConstants.FILE_PATH));
}
searcher.close();
}
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running the indexing program during chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory, index directory and indexes, you are ready for this step which is compiling and running your program. To do this, Keep `LuceneTester.java` file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

10 documents found. Time :31ms
Score: 1.3179655 File: E:\Lucene\Data\record3.txt
Score: 0.790779 File: E:\Lucene\Data\record1.txt
Score: 0.790779 File: E:\Lucene\Data\record2.txt
Score: 0.790779 File: E:\Lucene\Data\record4.txt
Score: 0.790779 File: E:\Lucene\Data\record5.txt
Score: 0.790779 File: E:\Lucene\Data\record6.txt
Score: 0.790779 File: E:\Lucene\Data\record7.txt
Score: 0.790779 File: E:\Lucene\Data\record8.txt
Score: 0.790779 File: E:\Lucene\Data\record9.txt
Score: 0.2635932 File: E:\Lucene\Data\record10.txt
10 documents found. Time :0ms
Score: 0.790779 File: E:\Lucene\Data\record1.txt
Score: 0.2635932 File: E:\Lucene\Data\record10.txt
Score: 0.790779 File: E:\Lucene\Data\record2.txt
Score: 1.3179655 File: E:\Lucene\Data\record3.txt
Score: 0.790779 File: E:\Lucene\Data\record4.txt
Score: 0.790779 File: E:\Lucene\Data\record5.txt
Score: 0.790779 File: E:\Lucene\Data\record6.txt
Score: 0.790779 File: E:\Lucene\Data\record7.txt
Score: 0.790779 File: E:\Lucene\Data\record8.txt
Score: 0.790779 File: E:\Lucene\Data\record9.txt

```