LUCENE - QUICK GUIDE

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	**Acquire Raw Content**	First step of any search application is to collect the target contents on which search are to be conducted.
2	**Build the document**	Next step is to build the document*s* from the raw contents which search application can understands and interpret easily.
3	**Analyze the document**	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	**Indexing the document**	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	**User Interface for Search**	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	**Build Query**	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	**Search Query**	Using query object, index database is then checked to get the relevant details and the content documents.
8	**Render Results**	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 on.

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 - ENVIRONMENT SETUP

Environment Setup

This tutorial will guide you on how to prepare a development environment to start your work with Spring Framework. This tutorial will also teach you how to setup JDK, Tomcat and Eclipse on your machine before you setup Spring Framework:

Step 1 - Setup Java Development Kit *JDK*:

You can download the latest version of SDK from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

If you are running Windows and installed the JDK in C:\jdk1.6.0_15, you would have to put the following line in your C:\autoexec.bat file.

```
set PATH=C:\jdk1.6.0_15\bin;%PATH%
set JAVA_HOME=C:\jdk1.6.0_15
```

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix *Solaris, Linux, etc.* , if the SDK is installed in /usr/local/jdk1.6.0_15 and you use the C shell, you would put the following into your .cshrc file.

```
setenv PATH /usr/local/jdk1.6.0_15/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.6.0_15
```

Alternatively, if you use an Integrated Development Environment *IDE* like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java, otherwise do proper setup as given document of the IDE.

Step 2 - Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So I would suggest you should have latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from http://www.eclipse.org/downloads/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\eclipse on windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

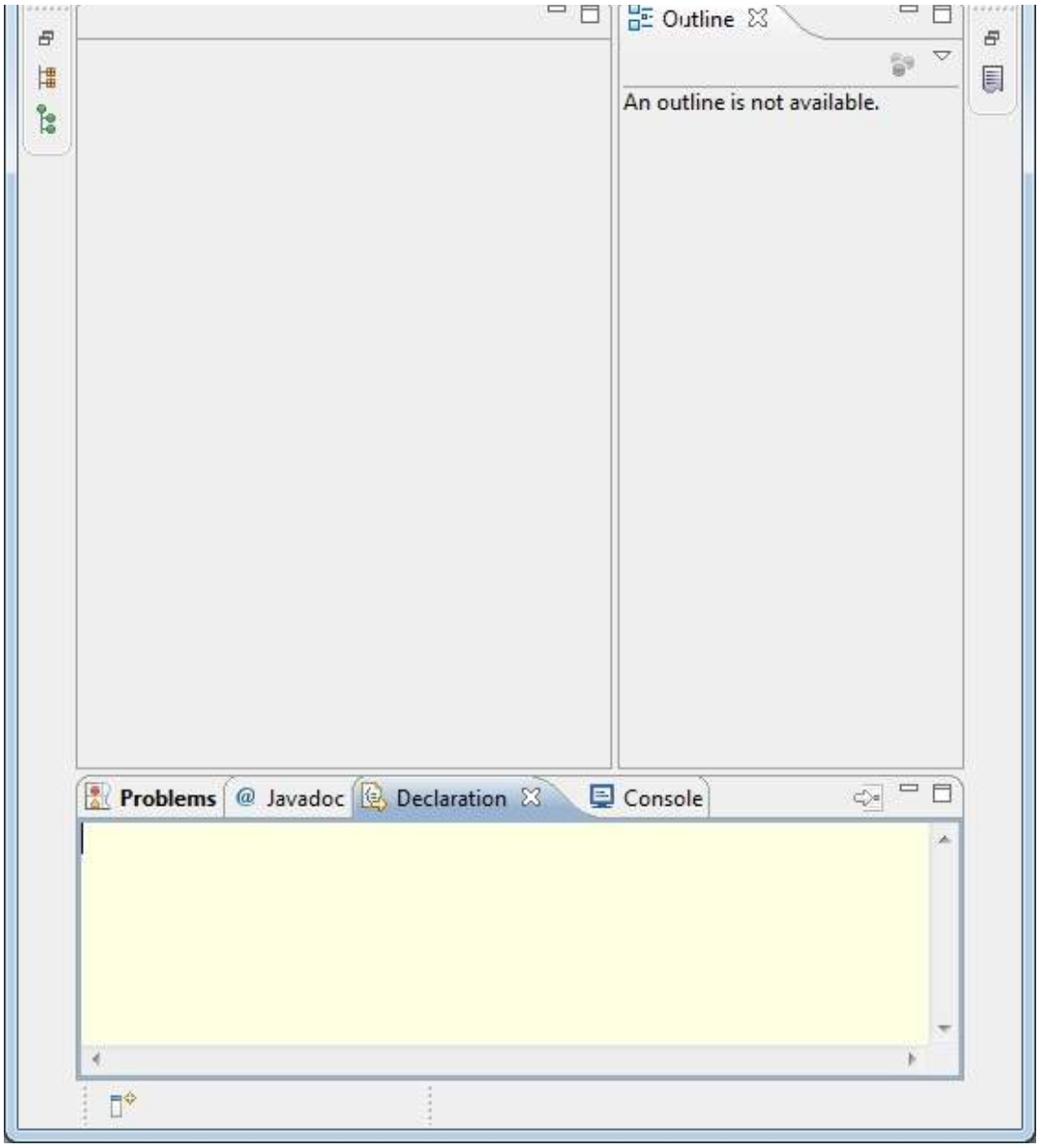
Eclipse can be started by executing the following commands on windows machine, or you can simply double click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix *Solaris, Linux, etc.* machine:

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display following result:

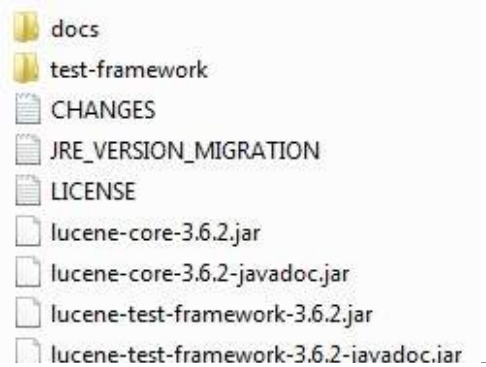


Step 3 - Setup Lucene Framework Libraries

Now if everything is fine, then you can proceed to setup your Lucene framework. Following are the simple steps to download and install the framework on your machine.

http://archive.apache.org/dist/lucene/java/3.6.2/

- Make a choice whether you want to install Lucene on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.
- Download the suitable version of Lucene framework binaries from http://archive.apache.org/dist/lucene/java/.
- At the time of writing this tutorial, I downloaded **lucene-3.6.2.zip** on my Windows machine and when you unzip the downloaded file it will give you directory structure inside C:\lucene-3.6.2 as follows.



You will find all the Lucene libraries in the directory **C:\lucene-3.6.2**. Make sure you set your CLASSPATH variable on this directory properly otherwise you will face problem while running your application. If you are using Eclipse then it is not required to set CLASSPATH because all the setting will be done through Eclipse.

Once you are done with this last step, you are ready to proceed for your first Lucene Example which you will see in the next chapter.

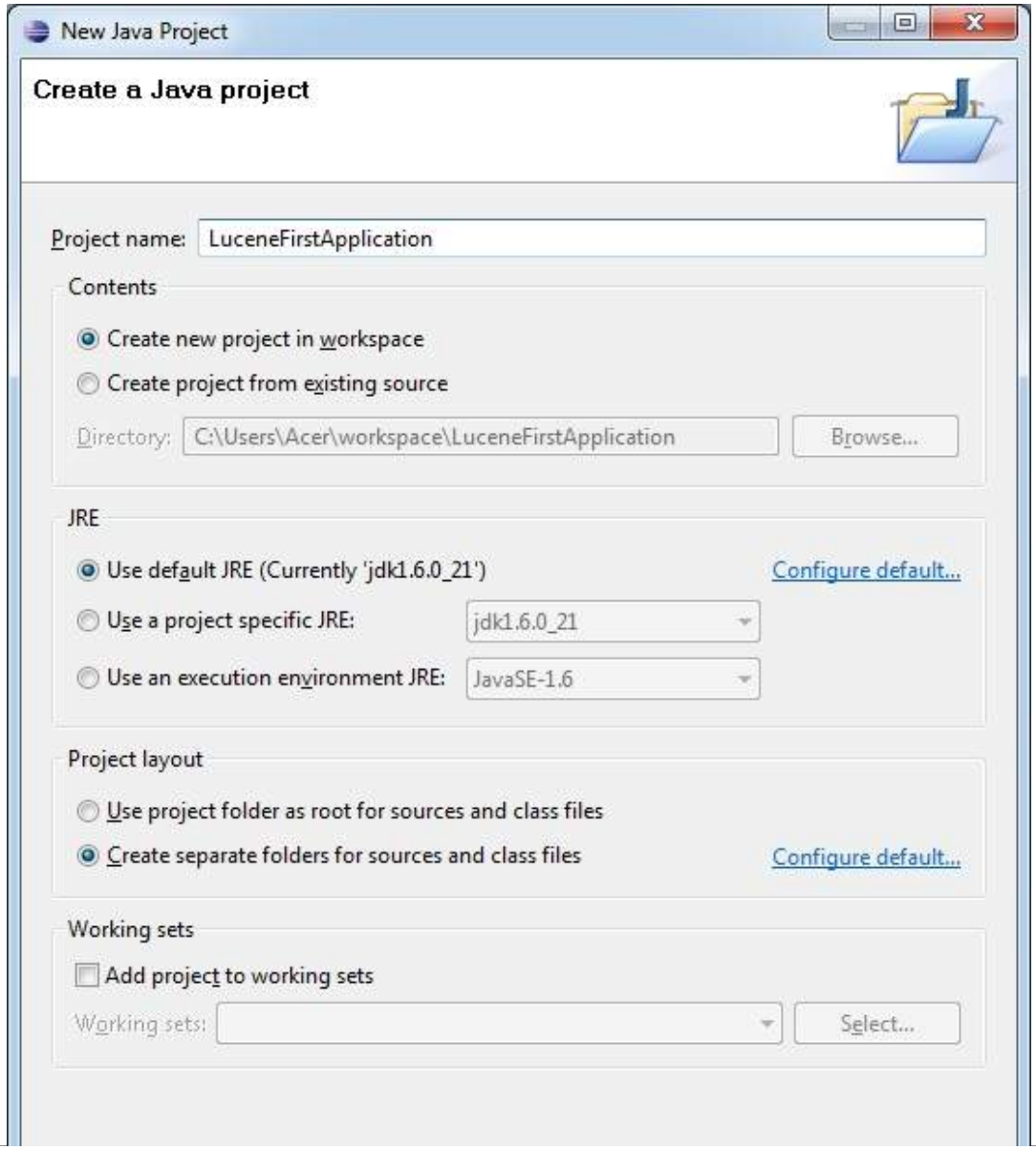
LUCENE - FIRST APPLICATION

Let us start actual programming with Lucene Framework. Before you start writing your first example using Lucene framework, you have to make sure that you have setup your Lucene environment properly as explained in Lucene - Environment Setup tutorial. I also assume that you have a little bit working knowledge with Eclipse IDE.

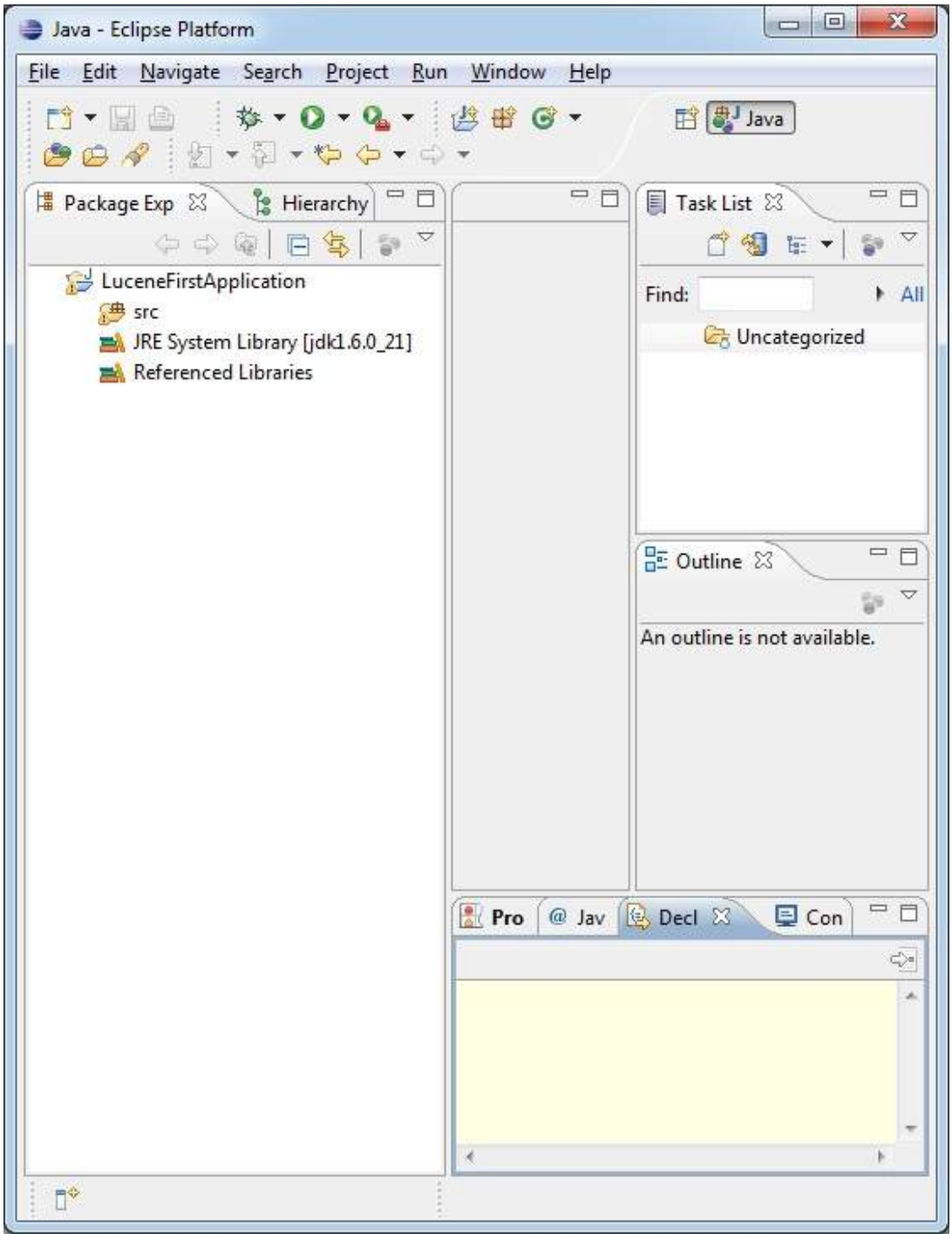
So let us proceed to write a simple Search Application which will print number of search result found. We'll also see the list of indexes created during this process.

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:

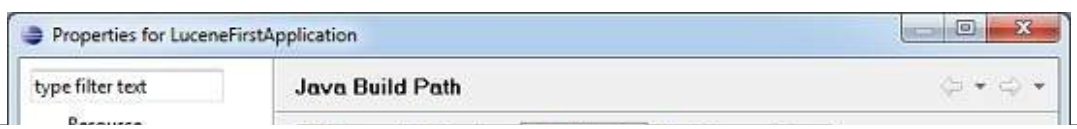


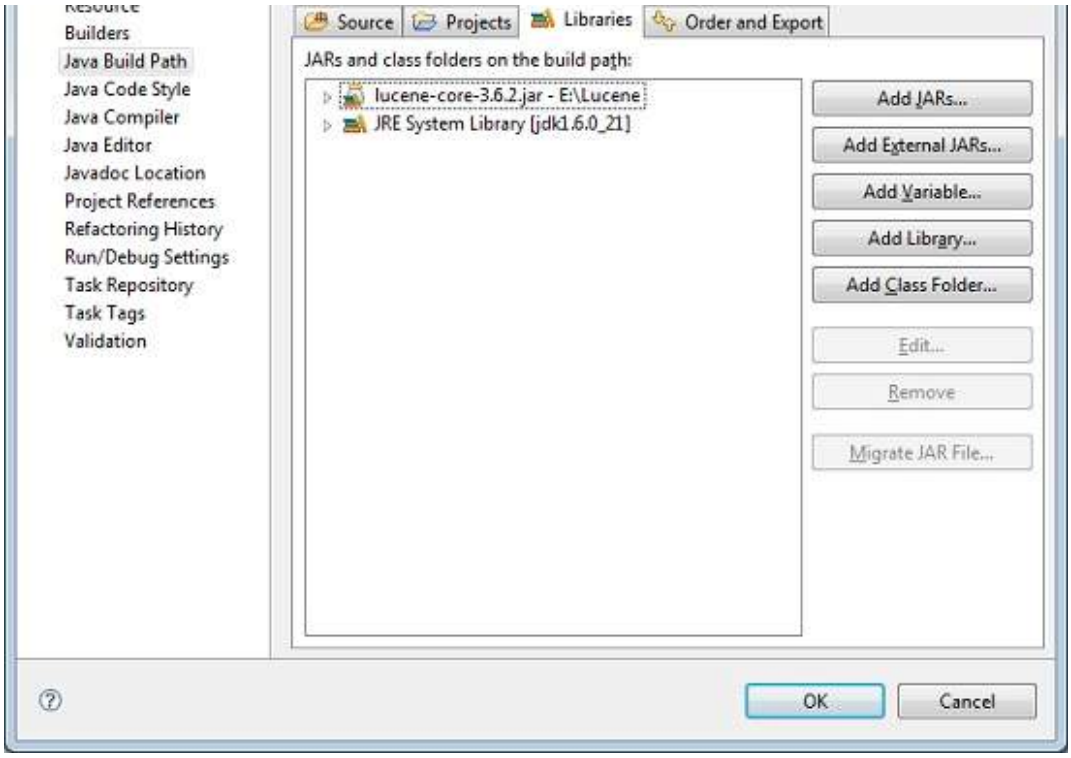
Once your project is created successfully, you will have following content in your **Project Explorer:**



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.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;
   }

   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.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 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.

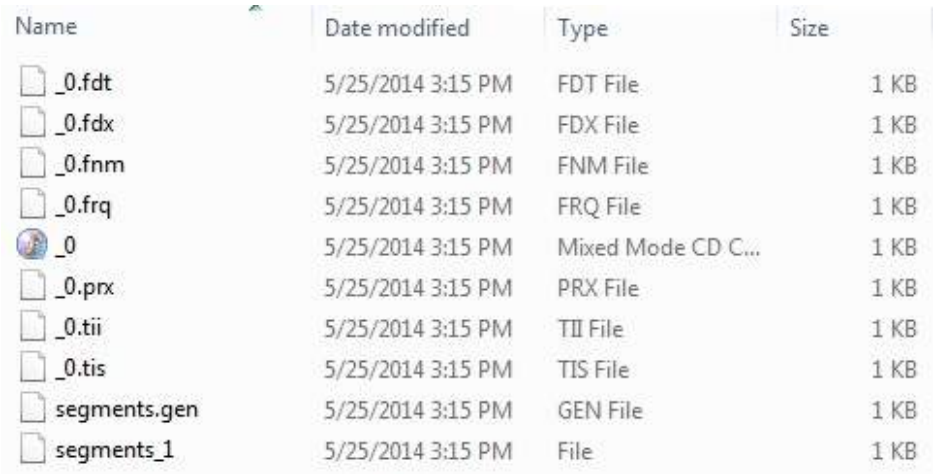
Step 5 - Running the Program:

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:

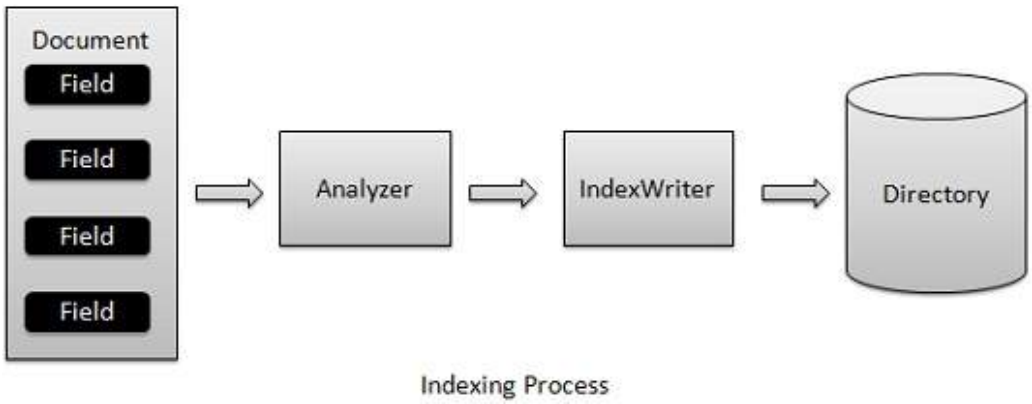
```
Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing 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:**



LUCENE - INDEXING CLASSES

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 *Documents* containing *Fields* to *IndexWriter* which analyzes the *Documents* 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.	Class & Description

No.	
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 is 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** *Directoryd, Analyzera, booleancreate, IndexDeletionPolicydeletionPolicy, IndexWriter. MaxFieldLengthmfl* Deprecated. use IndexWriter*Directory, IndexWriterConfig* instead.
2	**IndexWriter***Directoryd, Analyzera, booleancreate, IndexWriter. MaxFieldLengthmfl* Deprecated. use IndexWriter*Directory, IndexWriterConfig* instead.
3	**IndexWriter***Directoryd, Analyzera, IndexDeletionPolicydeletionPolicy, IndexWriter. MaxFieldLengthmfl* Deprecated. use IndexWriter*Directory, IndexWriterConfig* instead.
4	**IndexWriter** *Directoryd, Analyzera, IndexDeletionPolicydeletionPolicy, IndexWriter. MaxFieldLengthmfl, IndexCommitcommit* Deprecated. use IndexWriter*Directory, IndexWriterConfig* instead.
5	**IndexWriter***Directoryd, Analyzera, IndexWriter. MaxFieldLengthmfl* Deprecated. use IndexWriter*Directory, IndexWriterConfig* instead.
6	**IndexWriter***Directoryd, IndexWriterConfigconf* Constructs a new IndexWriter per the settings given in conf.

Class methods

S.N.	Method & Description
1	**void addDocument***Documentdoc* Adds a document to this index.
2	**void addDocument***Documentdoc, Analyzeranalyzer* Adds a document to this index, using the provided analyzer instead of the value of getAnalyzer.

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, Analyzeranalyzer*

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 addIndexes*Directory. . .* instead.

8

void close

Commits all changes to an index and closes all associated files.

9

void close*booleanwaitForMerges*

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*Queryquery*

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*Termterm*

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 *newsegments*$_N$*filewritten*.

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*booleanincludePendingClose*

Used internally to throw an AlreadyClosedException if this IndexWriter has been closed.

22

void expungeDeletes

Deprecated.

23

void expungeDeletes*booleandoWait*

Deprecated.

24

protected void flush*booleantriggerMerge, booleanapplyAllDeletes*

Flush all in-memory buffered updates *addsanddeletes* to the Directory.

25

protected void flush*booleantriggerMerge, booleanflushDocStores, booleanflushDeletes*

NOTE: flushDocStores is ignored now *hardwiredtotrue*; this method is only here for backwards compatibility.

26

void forceMerge*intmaxNumSegments*

Forces merge policy to merge segments until there's <= maxNumSegments.

27

void forceMerge*intmaxNumSegments, booleandoWait*

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*booleandoWait*

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*inttermInfosIndexDivisor*

Deprecated.Please use IndexReader.open*IndexWriter, boolean* instead. Furthermore, this method cannot guarantee the reader *anditssub − 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*Directorydirectory*

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 *stillintheRAMbuffer*, 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 *testagainifmergesareneeded* until no more merges are returned by the mergePolicy.

59

void merge*MergePolicy. OneMergemerge*

Merges the indicated segments, replacing them in the stack with a single segment.

60

void message*Stringmessage*

Prints a message to the infoStream *ifnon − null*, prefixed with the identifying information for

this writer and the thread that's calling it.

61

int numDeletedDocs*SegmentInfoinfo*

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 *stillintheRAMbuffer*, and including deletions.

63

int numRamDocs

Expert: Return the number of documents currently buffered in RAM.

64

void optimize

Deprecated.

65

void optimize*booleandoWait*

Deprecated.

66

void optimize*intmaxNumSegments*

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 *orsinceitwasopened, ifcommithasn't beencalled*.

71

String segString

72

String segString*Iterable < SegmentInfo > infos*

73

String segString*SegmentInfoinfo*

74

static void setDefaultInfoStream*PrintStreaminfoStream*

If non-null, this will be the default infoStream used by a newly instantiated IndexWriter.

75

static void setDefaultWriteLockTimeout*longwriteLockTimeout*

Deprecated.use IndexWriterConfig.setDefaultWriteLockTimeout*long* instead

76

void setInfoStream*PrintStreaminfoStream*

If non-null, information about merges, deletes and a message when maxFieldLength is reached will be printed to this.

77

void setMaxBufferedDeleteTerms*intmaxBufferedDeleteTerms*

Deprecated.use IndexWriterConfig.setMaxBufferedDeleteTerms*int* instead.

78

void setMaxBufferedDocs*intmaxBufferedDocs*

Deprecated.use IndexWriterConfig.setMaxBufferedDocs*int* instead.

79

void setMaxFieldLength*intmaxFieldLength*

Deprecated.use LimitTokenCountAnalyzer instead. Note that the behvaior 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*intmaxMergeDocs*

Deprecated.use LogMergePolicy.setMaxMergeDocs*int* directly.

81

void setMergedSegmentWarmer*IndexWriter. IndexReaderWarmerwarmer*

Deprecated.use IndexWriterConfig.setMergedSegmentWarmer *org. apache. lucene. index. IndexWriter. IndexReaderWarmer* instead.

82

void setMergeFactor*intmergeFactor*

Deprecated.use LogMergePolicy.setMergeFactor*int* directly.

83

void setMergePolicy*MergePolicymp*

Deprecated.use IndexWriterConfig.setMergePolicy*MergePolicy* instead.

84

void setMergeScheduler*MergeSchedulermergeScheduler*

Deprecated.use IndexWriterConfig.setMergeScheduler*MergeScheduler* instead

85

void setPayloadProcessorProvider*PayloadProcessorProviderpcp*

Sets the PayloadProcessorProvider to use when merging payloads.

86

void setRAMBufferSizeMB*doublemb*

Deprecated.use IndexWriterConfig.setRAMBufferSizeMB*double* instead.

87

void setReaderTermsIndexDivisor*intdivisor*

Deprecated.use IndexWriterConfig.setReaderTermsIndexDivisor*int* instead.

88

void setSimilarity*Similaritysimilarity*

Deprecated.use IndexWriterConfig.setSimilarity*Similarity* instead

89

void setTermIndexInterval*intinterval*

Deprecated.use IndexWriterConfig.setTermIndexInterval*int*

90

void setUseCompoundFile*booleanvalue*

Deprecated.use LogMergePolicy.setUseCompoundFile*boolean*.

91

void setWriteLockTimeout*longwriteLockTimeout*

Deprecated.use IndexWriterConfig.setWriteLockTimeout*long* instead

92

static void unlock*Directorydirectory*

Forcibly unlocks the index in the named directory.

93

void updateDocument*Termterm, Documentdoc*

Updates a document by first deleting the document*s* containing term and then adding the new document.

94

void updateDocument*Termterm, Documentdoc, Analyzeranalyzer*

Updates a document by first deleting the document*s* containing term and then adding the new document.

95

void updateDocuments*TermdelTerm, 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*TermdelTerm, Collection < Document > docs, Analyzeranalyzer*

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 *implementslockingforthisDirectoryinstance*.

Class constructors

S.N.	Constructor & Description
1	**Directory**

Class methods

S.N.	Method & Description
1	

void clearLock*Stringname*

Attempt to clear *forcefullyunlockandremove* the specified lock.

2

abstract void close

Closes the store.

3

static void copy*Directorysrc, Directorydest, booleancloseDirSrc*

Deprecated. Should be replaced with calls to copy*Directory*, *String*, *String* for every file that needs copying. You can use the following code:

```
IndexFileNameFilter filter = IndexFileNameFilter.getFilter();
for (String file : src.listAll()) {
  if (filter.accept(null, file)) {
    src.copy(dest, file, file);
  }
}
```

4

void copy*Directoryto, Stringsrc, Stringdest*

Copies the file src to Directory to under the new file name dest.

5

abstract IndexOutput createOutput*Stringname*

Creates a new, empty file in the directory with the given name.

6

abstract void deleteFile*Stringname*

Removes an existing file in the directory.

7

protected void ensureOpen

8

abstract boolean fileExists*Stringname*

Returns true iff a file with the given name exists.

9

abstract long fileLength*Stringname*

Returns the length of a file in the directory.

10

abstract long fileModified*Stringname*

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*Stringname*

Construct a Lock.

15

abstract IndexInput openInput*Stringname*

Returns a stream reading an existing file.

16

IndexInput openInput*Stringname, intbufferSize*

Returns a stream reading an existing file, with the specified read buffer size.

17

void setLockFactory*LockFactorylockFactory*

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*Stringname*

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*Stringname*

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***Fieldablefield* Just like getPositionIncrementGap*java. lang. String*, except for Token offsets instead.
3	**int getPositionIncrementGap***StringfieldName* 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***StringfieldName, Readerreader* 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***Objectobj* Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the same thread.
7	**abstract TokenStream tokenStream***StringfieldName, Readerreader* 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*Stringname*** Attempt to clear *forcefullyunlockandremove* the specified lock.
2	**void add*Fieldablefield*** Adds a field to a document.
3	**String get*Stringname*** Returns the string value of the field with the given name if any exist in this document, or null.
4	**byte[] getBinaryValue*Stringname*** Returns an array of bytes for the first *oronly* field that has the name specified as the method parameter.
5	

byte[][] getBinaryValues*Stringname*

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*Stringname*

Deprecated. Use getFieldable*java. lang. String* instead and cast depending on data type.

8

Fieldable getFieldable*Stringname*

Returns a field with the given name if any exist in this document, or null.

9

Fieldable[] getFieldables*Stringname*

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*Stringname*

Deprecated. Use getFieldable*java. lang. String* instead and cast depending on data type.

12

String[] getValues*Stringname*

Returns an array of values of the field specified as the method parameter.

13

void removeField*Stringname*

Removes field with the specified name from the document.

14

void removeFields*Stringname*

Removes all fields with the given name from the document.

15

void setBoost*floatboost*

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** ***Stringname, booleaninternName, Stringvalue, Field. Storestore, Field. Indexindex, Field. TermVectortermVector*** Create a field by specifying its name, value and how it will be saved in the index.
2	**Field*****Stringname, byte*[]*value*** Create a stored field with binary value.
3	**Field*****Stringname, byte*[]*value, Field. Storestore*** Deprecated.
4	**Field*****Stringname, byte*[]*value, intoffset, intlength*** Create a stored field with binary value.
5	**Field*****Stringname, byte*[]*value, intoffset, intlength, Field. Storestore*** Deprecated.
6	**Field*****Stringname, Readerreader*** Create a tokenized and indexed field that is not stored.

7

Field*Stringname, Readerreader, Field. TermVectortermVector*

Create a tokenized and indexed field that is not stored, optionally with storing term vectors.

8

Field*Stringname, Stringvalue, Field. Storestore, Field. Indexindex*

Create a field by specifying its name, value and how it will be saved in the index.

9

Field*Stringname, Stringvalue, Field. Storestore, Field. Indexindex, Field. TermVectortermVector*

Create a field by specifying its name, value and how it will be saved in the index.

10

Field*Stringname, TokenStreamtokenStream*

Create a tokenized and indexed field that is not stored.

11

Field*Stringname, TokenStreamtokenStream, Field. TermVectortermVector*

Create a tokenized and indexed field that is not stored, optionally with storing term vectors.

Class methods

S.N.	Method & Description

1

void clearLock*Stringname*

Attempt to clear *forcefullyunlockandremove* the specified lock.

2

Reader readerValue

The value of the field as a Reader, or null.

3

void setTokenStream*TokenStreamtokenStream*

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, intoffset, intlength*

Expert: change the value of this field.

6

void setValue*Readervalue*

Expert: change the value of this field.

7	**void setValue***Stringvalue* Expert: change the value of this field.
8	**String stringValue** The value of the field as a String, or null.
9	**TokenStream tokenStreamValue** The TokesStream 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 queries 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***Directorypath* Deprecated. Use IndexSearcher*IndexReader* instead.
2	**IndexSearcher***Directorypath, booleanreadOnly* Deprecated. Use IndexSearcher*IndexReader* instead.
3	**IndexSearcher***IndexReaderr* Creates a searcher searching the provided index.
4	**IndexSearcher***IndexReaderr, ExecutorServiceexecutor* Runs searches for each segment separately, using the provided ExecutorService.
5	**IndexSearcher***IndexReaderreader, IndexReader[]subReaders, int[]docStarts* Expert: directly specify the reader, subReaders and their docID starts.
6	**IndexSearcher***IndexReaderreader, IndexReader[]subReaders, int[]docStarts, ExecutorServiceexecutor* 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*IndexReaderr*.
2	**Weight createNormalizedWeight*Queryquery*** Creates a normalized weight for a top-level Query.
3	**Document doc*intdocID*** Returns the stored fields of document i.
4	**Document doc*intdocID, FieldSelectorfieldSelector*** Get the Document at the nth position.
5	**int docFreq*Termterm*** Returns total docFreq for this term.
6	**Explanation explain*Queryquery, intdoc*** Returns an Explanation that describes how doc scored against query.
7	**Explanation explain*Weightweight, intdoc*** 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*Queryoriginal*

Expert: called to re-write queries into primitive queries.

14

void search*Queryquery, Collectorresults*

Lower-level search API.

15

void search*Queryquery, Filterfilter, Collectorresults*

Lower-level search API.

16

TopDocs search*Queryquery, Filterfilter, intn*

Finds the top n hits for query, applying filter if non-null.

17

TopFieldDocs search*Queryquery, Filterfilter, intn, Sortsort*

Search implementation with arbitrary sorting.

18

TopDocs search*Queryquery, intn*

Finds the top n hits for query.

19

TopFieldDocs search*Queryquery, intn, Sortsort*

Search implementation with arbitrary sorting and no filter.

20

void search*Weightweight, Filterfilter, Collectorcollector*

Lower-level search API.

21

TopDocs search*Weightweight, Filterfilter, intnDocs*

Expert: Low-level search implementation.

22

TopFieldDocs search*Weightweight, Filterfilter, intnDocs, Sortsort*

Expert: Low-level search implementation with arbitrary sorting.

23

protected TopFieldDocs search*Weightweight, Filterfilter, intnDocs, Sortsort, booleanfillFields*

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*Weightweight, Filterfilter, ScoreDocafter, intnDocs*

Expert: Low-level search implementation.

25

TopDocs searchAfter*ScoreDocafter, Queryquery, Filterfilter, intn*

Finds the top n hits for query, applying filter if non-null, where all results are after a previous result *after*.

26

TopDocs searchAfter*ScoreDocafter, Queryquery, intn*

Finds the top n hits for query where all results are after a previous result *after*.

27

void setDefaultFieldSortScoring*booleandoTrackScores, booleandoMaxScore*

By default, no scores are computed when sorting by field *usingsearch*(*Query, Filter, int, Sort*).

28

void setSimilarity*Similaritysimilarity*

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*Stringfld*

Constructs a Term with the given field and empty text.

2

Term*Stringfld, Stringtxt*

Constructs a Term with the given field and text.

Class methods

S.N.	Method & Description
1	**void addDocument***Documentdoc* Adds a document to this index.
2	**int compareTo***Termother* 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***Stringtext* Optimized construction of new Terms by reusing same field as this Term - avoids field.intern overhead.
4	**boolean equals***Objectobj*
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***Searchersearcher* Expert: Constructs an appropriate Weight implementation for this query.
4	**boolean equals***Objectobj*
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***Searchersearcher* 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***IndexReaderreader* Expert: called to re-write queries into primitive queries.
11	**void setBoost***floatb* Sets the boost for this query clause to b.
12	**String toString** Prints a query to a string.
13	**abstract String toString***Stringfield* Prints a query to a string, with field assumed to be the default field and omitted.
14	**Weight weight***Searchersearcher* Deprecated. never ever use this method in Weight implementations. Subclasses of Query should use createWeight*org. apache. lucene. search. Searcher*, 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***Termt* Constructs a query for the term t.

Class methods

S.N.	Method & Description
1	**void addDocument***Documentdoc* Adds a document to this index.
2	**Weight createWeight***Searchersearcher* Expert: Constructs an appropriate Weight implementation for this query.
3	**boolean equals***Objecto* 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***Stringfield* 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***inttotalHits, ScoreDoc[]scoreDocs, floatmaxScore*

Class methods

S.N.	Method & Description
1	**getMaxScore** Returns the maximum score value encountered.
2	**static TopDocs merge***Sortsort, inttopN, TopDocs[]shardHits* Returns a new TopDocs, containing topN results across the provided TopDocs, sorting by the specified Sort.
3	**void setMaxScore***floatmaxScore* Sets the maximum score value encountered.

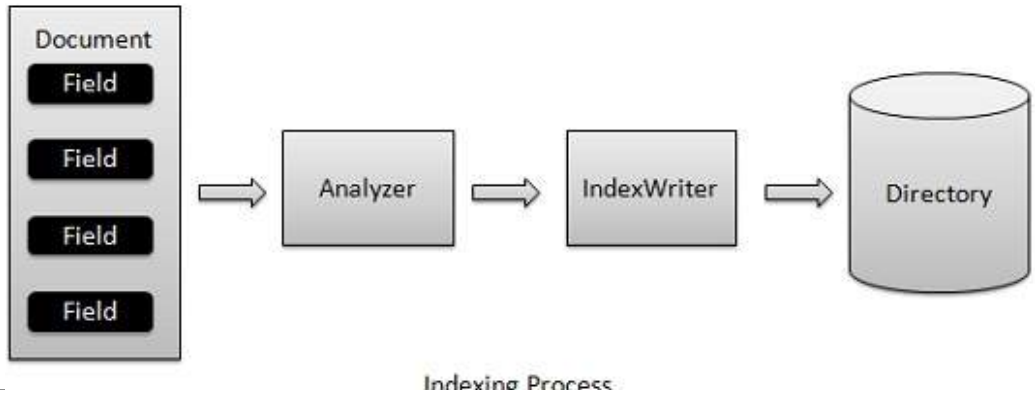
Methods inherited

This class inherits methods from the following classes:

- 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.



Indexing Process

We add *Documents* containing *Fields* to *IndexWriter* which analyzes the *Documents* 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.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;
}
```

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand indexing process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Indexer.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.

```java
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.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;
   }

   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.

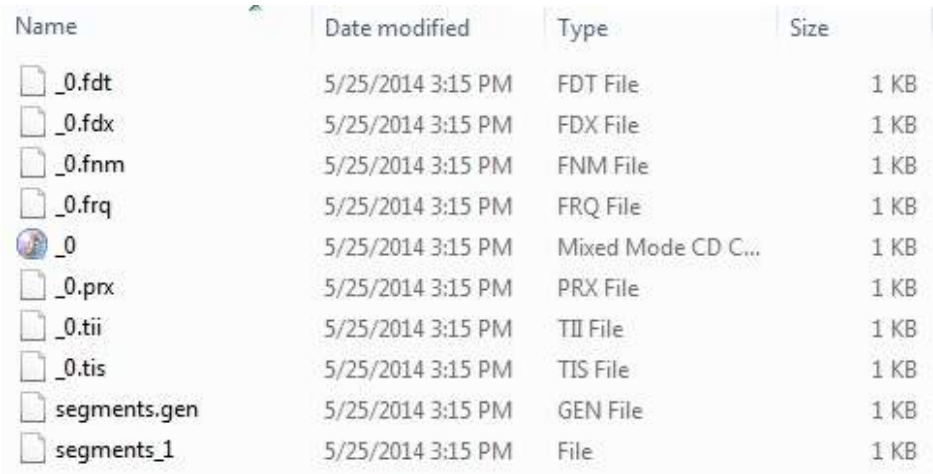
Running the Program:

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 - 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 proces 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 OPERATION

Add document is one of the core operation as part of indexing process.

We add *Documents* containing *Fields* 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.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;
}
```

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 IndexWricrter 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 *specifiedwhenindexwriteriscreated.*
- **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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *EJB - First Application* chapter as such for this chapter to understand indexing process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Indexer.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.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;
   }

   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.

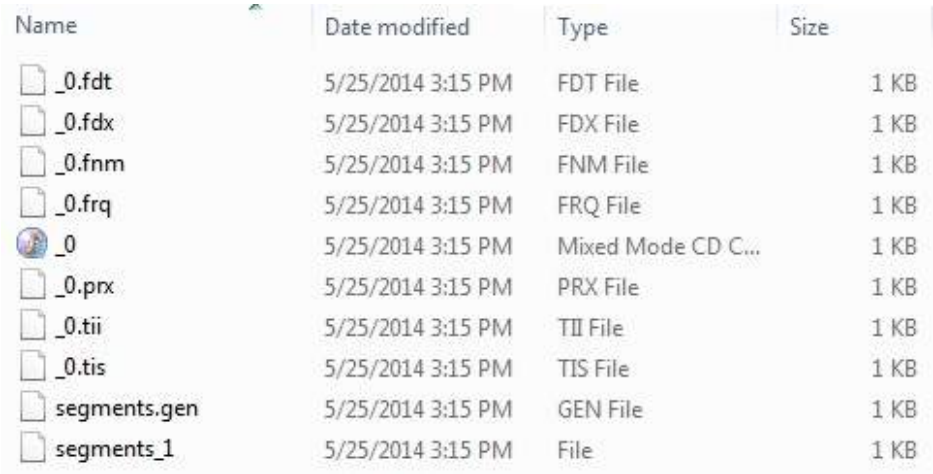
Running the Program:

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 OPERATION

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 *Documents* containing *Fields* 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 contents
   writer.updateDocument(
      new Term(LuceneConstants.CONTENTS,
      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 IndexWricrter 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 *specifiedwhenindexwriteriscreated.*
- **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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *EJB - First Application* chapter as such for this chapter to understand indexing process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Indexer.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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 contents
      writer.updateDocument(
         new Term(LuceneConstants.FILE_NAME,
         file.getName()),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();
   }
}
```

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.

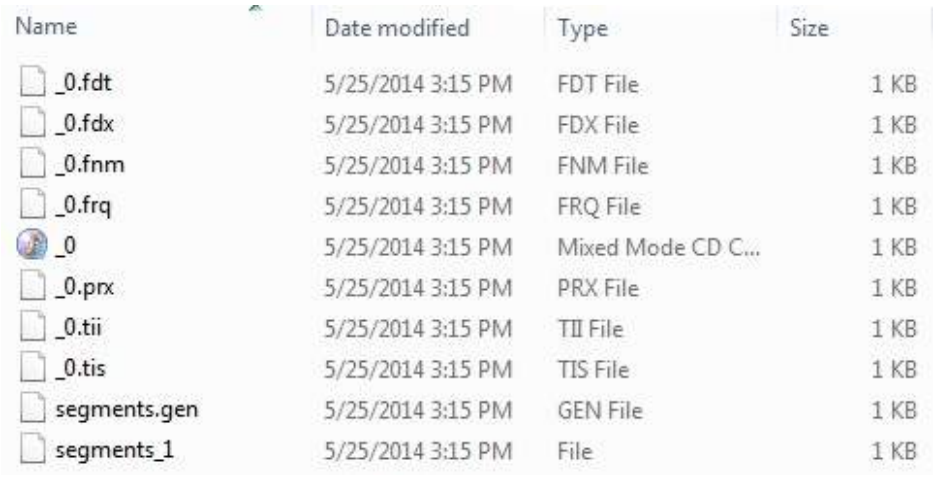
Running the Program:

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 OPERATION

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 *Documents* containing *Fields* 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();
   System.out.println("index contains deleted files: "+writer.hasDeletions());
   System.out.println("index contains documents: "+writer.maxDoc());
   System.out.println("index contains deleted documents: "+writer.numDoc());
}
```

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 IndexWricrter 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *EJB - First Application* chapter as such for this chapter to understand indexing process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Indexer.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.

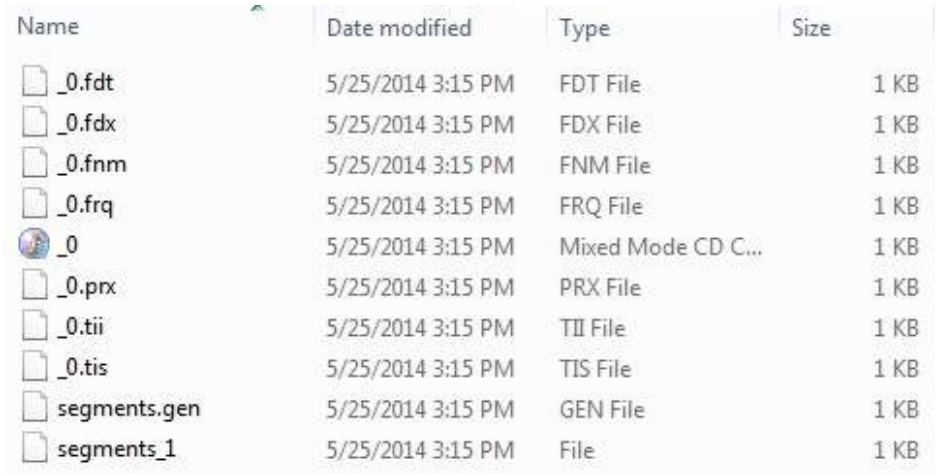
Running the Program:

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 *Documents* containing *Fields* 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 searcable sepearately.
- **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** - Varient of Index.ANALYZED. Analyzer will break the field's value into stream of tokens and each token is searcable sepearately 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** - Varient 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *EJB - First Application* chapter as such for this chapter to understand indexing process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Indexer.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.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;
   }

   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.

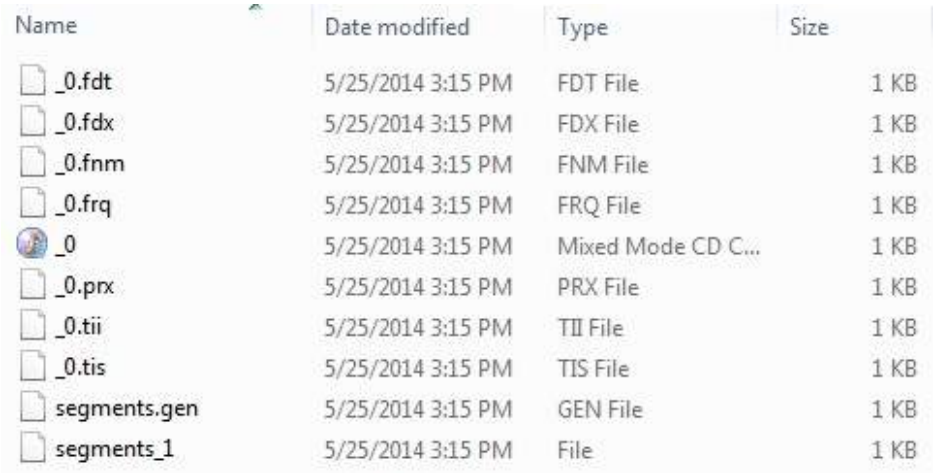
Running the Program:

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
```

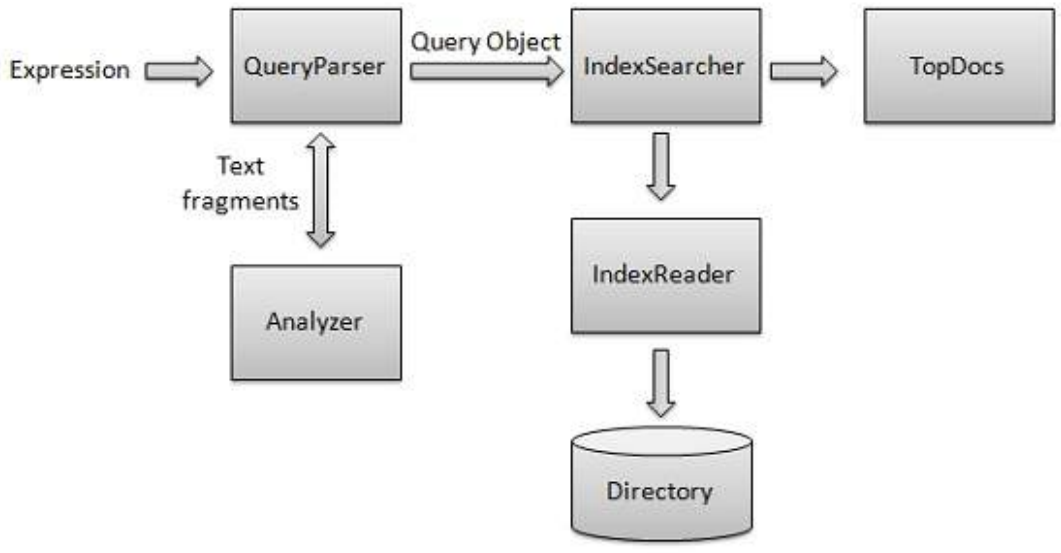
Once you've run the program successfully, you will have following content in your **index directory:**



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 OPERATION

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java*,*TextFileFilter.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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***Termt* Constructs a query for the term t.

Class methods

S.N.	Method & Description
1	**void addDocument***Documentdoc*

Adds a document to this index.

2

Weight createWeight*Searchersearcher*

Expert: Constructs an appropriate Weight implementation for this query.

3

boolean equals*Objecto*

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*Stringfield*

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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** ***Stringfield, StringlowerTerm, StringupperTerm, booleanincludeLower, booleanincludeUpper*** Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.
2	**TermRangeQuery** ***Stringfield, StringlowerTerm, StringupperTerm, booleanincludeLower, booleanincludeUpper, Collatorcollator*** Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.

Class methods

S.N.	Method & Description
1	

boolean equals*Objectobj*

2

Collator getCollator

Returns the collator used to determine range inclusion, if any.

3

protected FilteredTermEnum getEnum*IndexReaderreader*

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*Stringfield*

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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***Termprefix* Constructs a query for the term starting with prefix.

Class methods

S.N.	Method & Description
1	**boolean equals***Objecto* Returns true iff o is equal to this.
2	**protected FilteredTermEnumgetEnum***IndexReaderreader* 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***Stringfield* 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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***booleandisableCoord* Constructs an empty boolean query.

Class methods

S.N.	Method & Description

1

void add*BooleanClauseclause*

Adds a clause to a boolean query.

2

void add*Queryquery, BooleanClause. Occuroccur*

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*Searchersearcher*

Expert: Constructs an appropriate Weight implementation for this query.

6

boolean equals*Objecto*

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*IndexReaderreader*

Expert: called to re-write queries into primitive queries.

15

static void setMaxClauseCount*intmaxClauseCount*

Set the maximum number of clauses permitted per BooleanQuery.

16

void setMinimumNumberShouldMatch*intmin*

Specifies a minimum number of the optional BooleanClauses which must be satisfied.

17

String toString*Stringfield*

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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*Termterm*

Adds a term to the end of the query phrase.

2

void add*Termterm, intposition*

Adds a term to the end of the query phrase.

3

Weight createWeight*Searchersearcher*

Expert: Constructs an appropriate Weight implementation for this query.

4

boolean equals*Objecto*

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*IndexReaderreader*

Expert: called to re-write queries into primitive queries.

11

void setSlop*ints*

Sets the number of other words permitted between words in query phrase.

12

String toString*Stringf*

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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 - 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*Termterm*** Calls FuzzyQuery*term*, 0.5*f*, 0, *Integer*. MAX_VALUE.
2	**FuzzyQuery*Termterm, floatminimumSimilarity*** Calls FuzzyQuery*term, minimumSimilarity*, 0, *Integer*. MAX_VALUE.
3	**FuzzyQuery*Termterm, floatminimumSimilarity, intprefixLength*** Calls FuzzyQuery*term, minimumSimilarity, prefixLength, Integer*. MAX_VALUE.
4	**FuzzyQuery*Termterm, floatminimumSimilarity, intprefixLength, intmaxExpansions*** Create a new FuzzyQuery that will match terms with a similarity of at least minimum Similarity to term.

Class methods

1 **boolean equals*Objectobj***

2 **protected FilteredTermEnum getEnum*IndexReaderreader***

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 to String*Stringfield*.

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter

to understand searching process.

2 Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.

3 Create *LuceneTester.java* 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.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***StringnormsField*

Class methods

S.N.	Method & Description
1	**Weight createWeight***Searchersearcher* Expert: Constructs an appropriate Weight implementation for this query.
2	**boolean equals***Objecto*
3	

void extractTerms*Set < Term > terms*

Expert: adds all terms occurring in this query to the terms set.

4

int hashCode

5

String toString*Stringfield*

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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 *Documents* 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, startoffset, endoffset, tokentypeanditspositionincrement*.
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 spilts the text in a document based on whitespace.
5	SimpleAnalyzer This analyzer spilts 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 sofisticated 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, startoffset, endoffset, tokentypeanditspositionincrement*.

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 will null text.
2	**Token***char[]startTermBuffer, inttermBufferOffset, inttermBufferLength, intstart, intend* Constructs a Token with the given term buffer offset & length, start and end offsets.
3	**Token***intstart, intend* Constructs a Token with null text and start & end offsets.
4	**Token***intstart, intend, intflags* Constructs a Token with null text and start & end offsets plus flags.
5	**Token***intstart, intend, Stringtyp* Constructs a Token with null text and start & end offsets plus the Token type.
6	**Token***Stringtext, intstart, intend* Constructs a Token with the given term text, and start & end offsets.
7	**Token***Stringtext, intstart, intend, intflags* Constructs a Token with the given text, start and end offsets, & type.
8	**Token***Stringtext, intstart, intend, Stringtyp* 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, intnewTermOffset, intnewTermLength, intnewStartOffset, intnewEndOffset*** Makes a clone, but replaces the term buffer & start/end offset in the process.
4	**void copyTo***AttributeImpltarget* 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***Objectobj*
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*AttributeReflectorreflector*

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, intnewTermOffset, intnewTermLength, intnewStartOffset, intnewEndOffset

Shorthand for calling clear, CharTermAttributeImpl.copyBuffer*char*[], *int*, *int*, setStartOffset*int*, setEndOffset*int* setType*java. lang. String* on Token.DEFAULT_TYPE.

14

Token reinit
char*[]*newTermBuffer, intnewTermOffset, intnewTermLength, intnewStartOffset, intnewEndOffset, StringnewType

Shorthand for calling clear, CharTermAttributeImpl.copyBuffer*char*[], *int*, *int*, setStartOffset*int*, setEndOffset*int*, setType*java. lang. String*.

15

Token reinit*StringnewTerm, intnewStartOffset, intnewEndOffset*

Shorthand for calling clear, CharTermAttributeImpl.append*CharSequence*, setStartOffset*int*, setEndOffset*int* setType*java. lang. String* on Token.DEFAULT_TYPE.

16

Token reinit*StringnewTerm, intnewTermOffset, intnewTermLength, intnewStartOffset, intnewEndOffset*

Shorthand for calling clear, CharTermAttributeImpl.append*CharSequence, int, int*, setStartOffset*int* , set End Offset*int* setType*java. lang. String* on Token.DEFAULT_TYPE.

17

Token reinit
StringnewTerm, intnewTermOffset, intnewTermLength, intnewStartOffset, intnewEndOffset, StringnewType

Shorthand for calling clear, CharTermAttributeImpl.append*CharSequence, int, int*, setStartOffset*int* , setEndOffset*int* setType*java. lang. String*.

18

Token reinit*StringnewTerm, intnewStartOffset, intnewEndOffset, StringnewType*

Shorthand for calling clear, CharTermAttributeImpl.append*CharSequence*, setStartOffset*int*, setEndOffset*int* setType*java. lang. String*.

19

void reinit*Tokenprototype*

Copy the prototype token's fields into this one.

20

void reinit*Tokenprototype, char*[]*newTermBuffer, intoffset, intlength*

Copy the prototype token's fields into this one, with a different term.

21

void reinit*Tokenprototype, StringnewTerm*

Copy the prototype token's fields into this one, with a different term.

22

void setEndOffset*intoffset*

Set the ending offset.

23

void setFlags*intflags*

24

void setOffset*intstartOffset, intendOffset*

Set the starting and ending offset.

25

void setPayload*Payloadpayload*

Sets this Token's payload.

26

void setPositionIncrement*intpositionIncrement*

Set the position increment.

27

void setPositionLength*intpositionLength*

Set the position length.

28

void setStartOffset*intoffset*

Set the starting offset.

29

void setType*Stringtype*

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.TermAttributeImpl
- org.apache.lucene.analysis.tokenattributes.CharTermAttributeImpl
- org.apache.lucene.util.AttributeImpl
- java.lang.Object

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. AttributeFactoryfactory* A TokenStream using the supplied AttributeFactory for creating new Attribute instances.
3	**protected TokenStream***AttributeSourceinput* 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 *usingthenewTokenStreamAPI*.
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***Fieldablefield* Just like getPositionIncrementGap*java. lang. String*, except for Token offsets instead.
3	**int getPositionIncrementGap***StringfieldName* 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***StringfieldName, Readerreader* 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***Objectobj* Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the same thread.
7	**abstract TokenStream tokenStream***StringfieldName, Readerreader* 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 analyzer spilts the text in a document based on whitespace.

Class declaration

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

```
public final class WhitespaceAnalyzer
   extends ReusableAnalyzerBase
```

Class constructors

S.N.	Constructor & Description
1	**WhitespaceAnalyzer** Deprecated. use WhitespaceAnalyzer*Version* instead.
2	**White space Analyzer** ***VersionmatchVersion*** Creates a new WhitespaceAnalyzer.

Class methods

S.N.	Method & Description
1	**protected ReusableAnalyzerBase.TokenStreamComponents createComponents** ***StringfieldName, Readerreader*** 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 displayTokenUsingWhitespaceAnalyzer() throws IOException{
   String text = "Lucene is simple yet powerful java based search library.";
   Analyzer analyzer = new WhitespaceAnalyzer(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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.WhitespaceAnalyzer;
```

```
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.displayTokenUsingWhitespaceAnalyzer();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private void displayTokenUsingWhitespaceAnalyzer() throws IOException{
      String text
         = "Lucene is simple yet powerful java based search library.";
      Analyzer analyzer = new WhitespaceAnalyzer(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() + "] ");
      }
   }
}
```

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 - SIMPLEANALYZER

Introduction

This analyzer spilts 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*VersionmatchVersion***

Creates a new SimpleAnalyzer.

Class methods

S.N.	Method & Description
1	**protected ReusableAnalyzerBase.TokenStreamComponents createComponents** ***StringfieldName, Readerreader*** 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.CONTENTS, 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***VersionmatchVersion* Builds an analyzer which removes words in ENGLISH_STOP_WORDS_SET.
2	**StopAnalyzer***VersionmatchVersion, FilestopwordsFile* Builds an analyzer with the stop words from the given file.
3	**StopAnalyzer***VersionmatchVersion, Readerstopwords* Builds an analyzer with the stop words from the given reader.
4	**StopAnalyzer***VersionmatchVersion, Set < ? > stopWords* Builds an analyzer with the stop words from the given set.

Class methods

S.N.	Method & Description
1	**protected ReusableAnalyzerBase.TokenStreamComponents createComponents** ***StringfieldName, Readerreader*** 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.CONTENTS, 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 sofisticated 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***VersionmatchVersion* Builds an analyzer with the default stop words $STOP_WORDS_SET$.
2	**StandardAnalyzer***VersionmatchVersion, Filestopwords* Deprecated. Use StandardAnalyzer*Version, Reader* instead.

3

StandardAnalyzer*VersionmatchVersion, Readerstopwords*

Builds an analyzer with the stop words from the given reader.

4

StandardAnalyzer*VersionmatchVersion, Set < ? > stopWords*

Builds an analyzer with the given stop words.

Class methods

S.N.	Method & Description
1	**protected ReusableAnalyzerBase.TokenStreamComponents createComponents** *StringfieldName, Readerreader* Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.
2	**int getMaxTokenLength**
3	**void setMaxTokenLength***intlength* 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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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.CONTENTS, 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

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 *LuceneFirstApplication* under a package *com.tutorialspoint.lucene* as explained in the *Lucene - First Application* chapter. You can also use the project created in *Lucene - First Application* chapter as such for this chapter to understand searching process.
2	Create *LuceneConstants.java* and *Searcher.java* as explained in the *Lucene - First Application* chapter. Keep rest of the files unchanged.
3	Create *LuceneTester.java* 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
```