



Lucene

APACHE LUCENE search engine programming

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

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

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

Audience

This tutorial is designed for Software Professionals who are willing to learn Lucene search Engine Programming in simple and easy steps. After completing this tutorial, you will be at the intermediate level of expertise from where you can take yourself to a higher level of expertise.

Prerequisites

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

Copyright & Disclaimer

© Copyright 2016 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	1
Audience	1
Prerequisites	1
Copyright & Disclaimer.....	1
Table of Contents	2
1. LUCENE – OVERVIEW	5
How Search Application works?	5
Lucene's Role in Search Application	6
2. LUCENE – ENVIRONMENT SETUP	7
3. LUCENE – FIRST APPLICATION	10
4. LUCENE – INDEXING CLASSES	22
IndexWriter	23
Directory	34
Analyzer	37
Document	39
Field	41
5. LUCENE – SEARCHING CLASSES	45
Searching Classes	45
IndexSearcher	45
Term	50
Query	51
TermQuery	53
TopDocs	54

6.	LUCENE – INDEXING PROCESS.....	56
7.	LUCENE – INDEXING OPERATIONS.....	64
8.	LUCENE – SEARCH OPERATION.....	94
	Create a QueryParser	94
	Create a IndexSearcher	95
	Make search.....	95
	Get the Document.....	96
	Close IndexSearcher	96
9.	LUCENE – QUERY PROGRAMMING.....	101
	TermQuery.....	102
	TermRangeQuery	108
	PrefixQuery	115
	BooleanQuery	121
	PhraseQuery	130
	WildcardQuery.....	137
	FuzzyQuery	143
	MatchAllDocsQuery	150
10.	LUCENE – ANALYSIS.....	157
	Token	158
	TokenStream.....	162
	Analyzer	164
	WhitespaceAnalyzer	165
	SimpleAnalyzer	169
	StopAnalyzer	172
	StandardAnalyzer	176

11. LUCENE – SORTING	181
Sorting by Relevance	181
Sorting by IndexOrder	182
Data & Index Directory Creation	188

1. Lucene – Overview

Lucene is a simple yet powerful Java-based **Search** library. It can be used in any application to add search capability to it. Lucene is an open-source project. It is scalable. This high-performance library is 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?

A Search application performs all or a few of the following operations:

Step	Title	Description
1	Acquire Raw Content	The first step of any search application is to collect the target contents on which search application is to be conducted.
2	Build the document	The next step is to build the document(s) from the raw content, which the search application can understand and interpret easily.
3	Analyze the document	Before the indexing process starts, the document is to be analyzed as to which part of the text is a candidate to be indexed. This process is where the document is analyzed.
4	Indexing the document	Once documents are built and analyzed, the next step is to index them so that this document can be retrieved based on certain keys instead of the entire content 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 the application can make any search. To facilitate a user to make a search, the application must provide a user a mean or a user interface where a user can enter text and start the search process.
6	Build Query	Once a user makes a request to search a text, the 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 a query object, the index database is then checked to get the relevant details and the content documents.
8	Render Results	Once the result is received, the application should decide on 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, a search application can also provide **administration user interface** and help administrators of the application to control the level of search based on the user profiles. Analytics of search results 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 a nutshell, Lucene is the heart of any search application and provides vital operations pertaining to indexing and searching. Acquiring contents and displaying the results is left for the application part to handle.

In the next chapter, we will perform a simple Search application using Lucene Search library.

2. Lucene – Environment Setup

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

Step 1: Java Development Kit (JDK) Setup

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 the 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 in the document of the IDE.

Step 2: Eclipse IDE Setup

All the examples in this tutorial have been written using **Eclipse IDE**. So I would suggest you should have the 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.

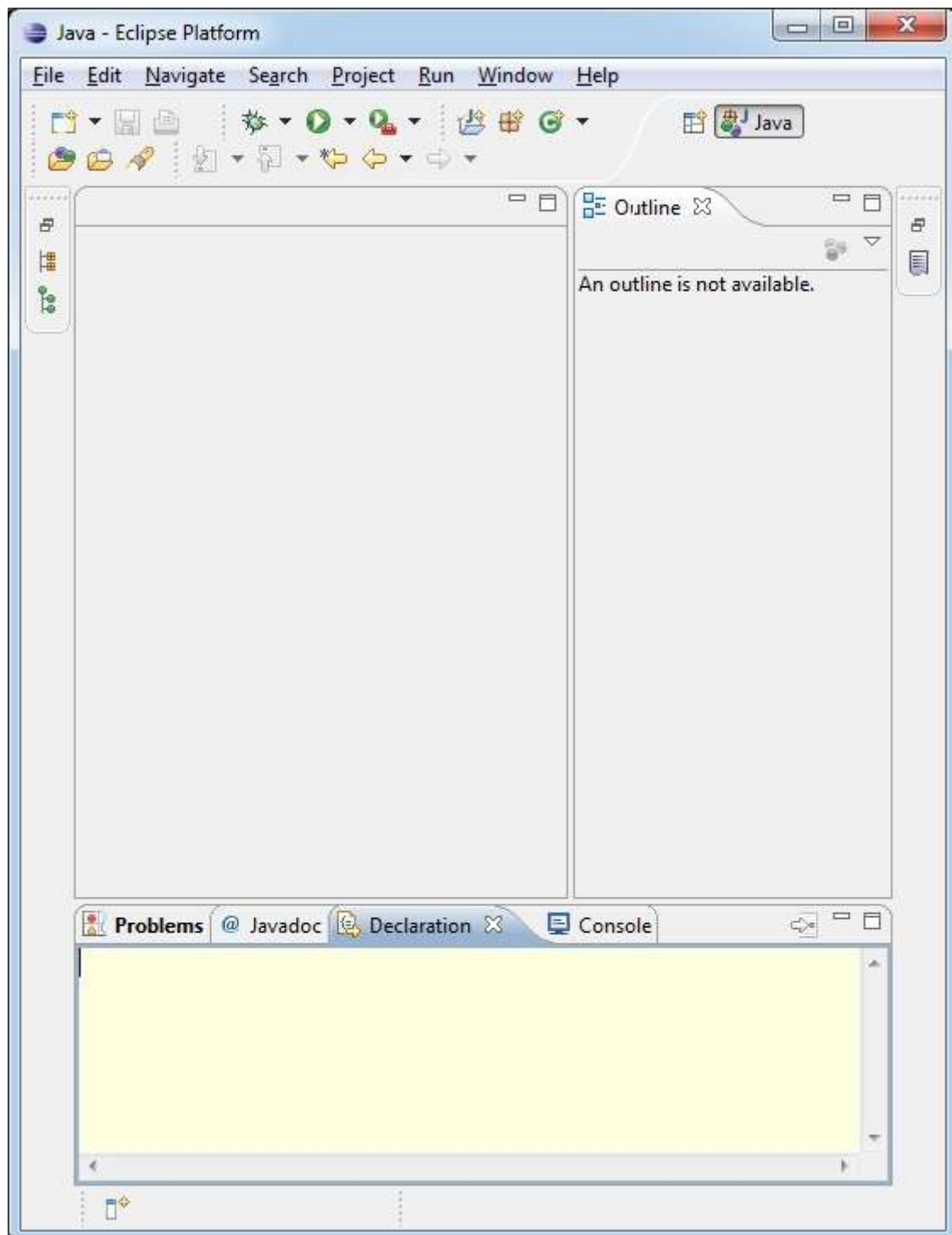
Eclipse can be started by executing the following commands on windows machine, or you can simply double click on **eclipse.exe**

```
%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, it should display the following result:

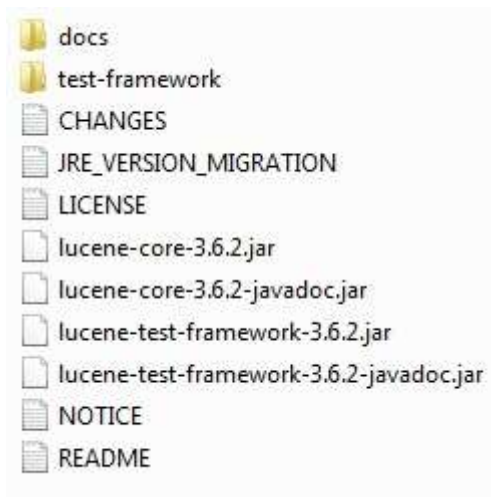


Step 3: Setup Lucene Framework Libraries

If the startup is successful, then you can proceed to set up 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 the .zip file for windows and .tar 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 the 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.

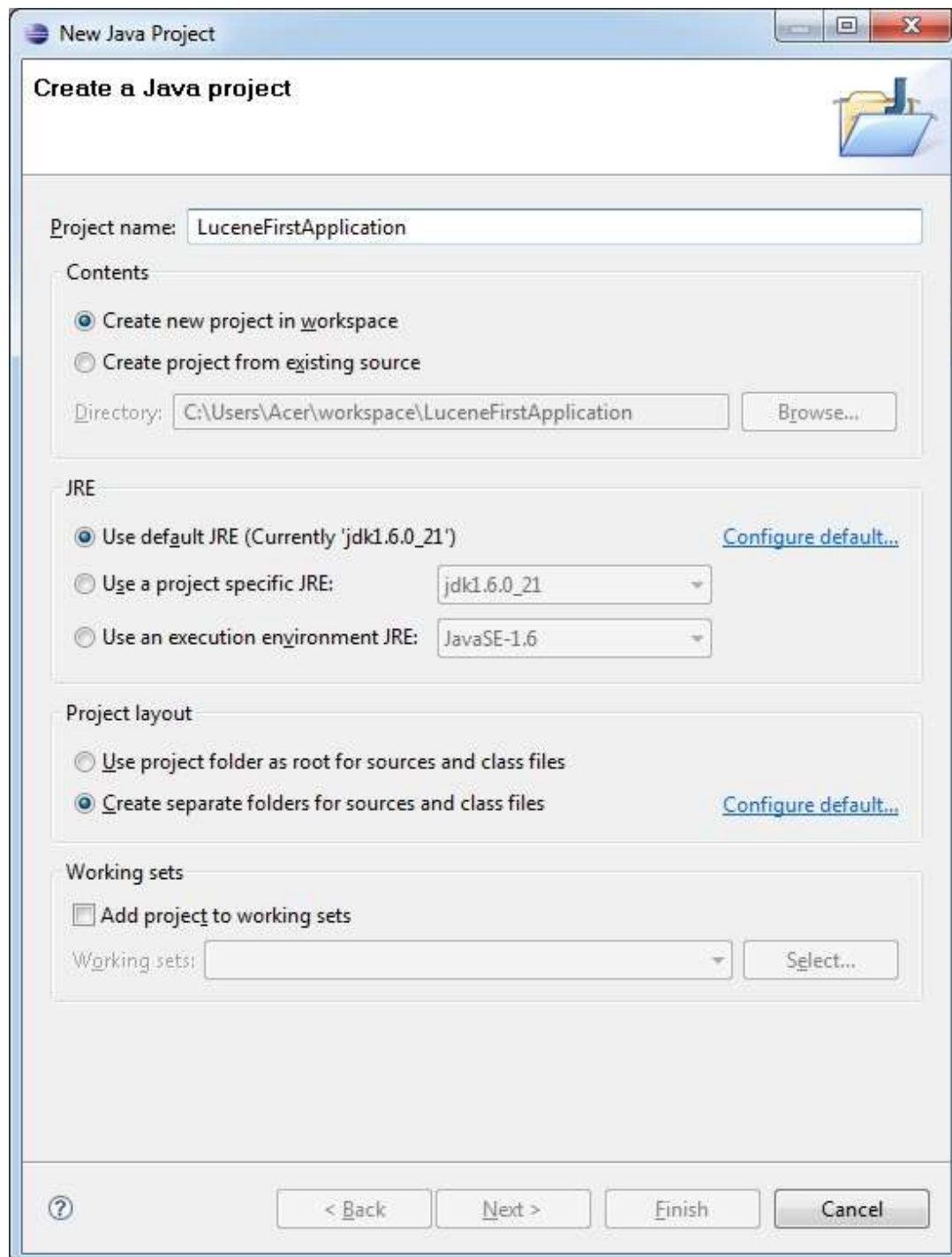
3. Lucene – First Application

In this chapter, we will learn the actual programming with Lucene Framework. Before you start writing your first example using Lucene framework, you have to make sure that you have set up your Lucene environment properly as explained in [Lucene - Environment Setup](#) tutorial. It is recommended you have the working knowledge of Eclipse IDE.

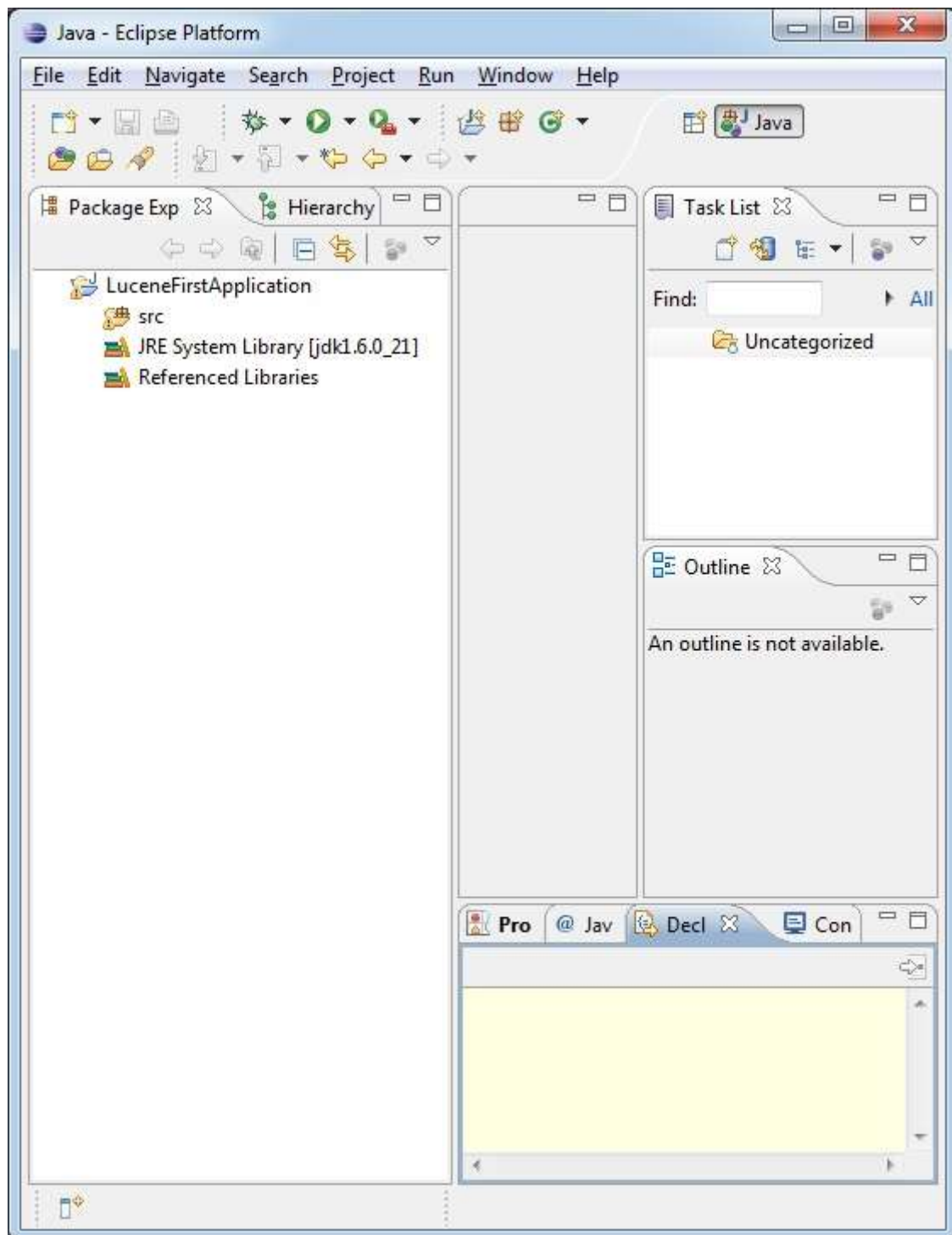
Let us now proceed by writing a simple Search Application which will print the number of search results found. We'll also see the list of indexes created during this process.

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:

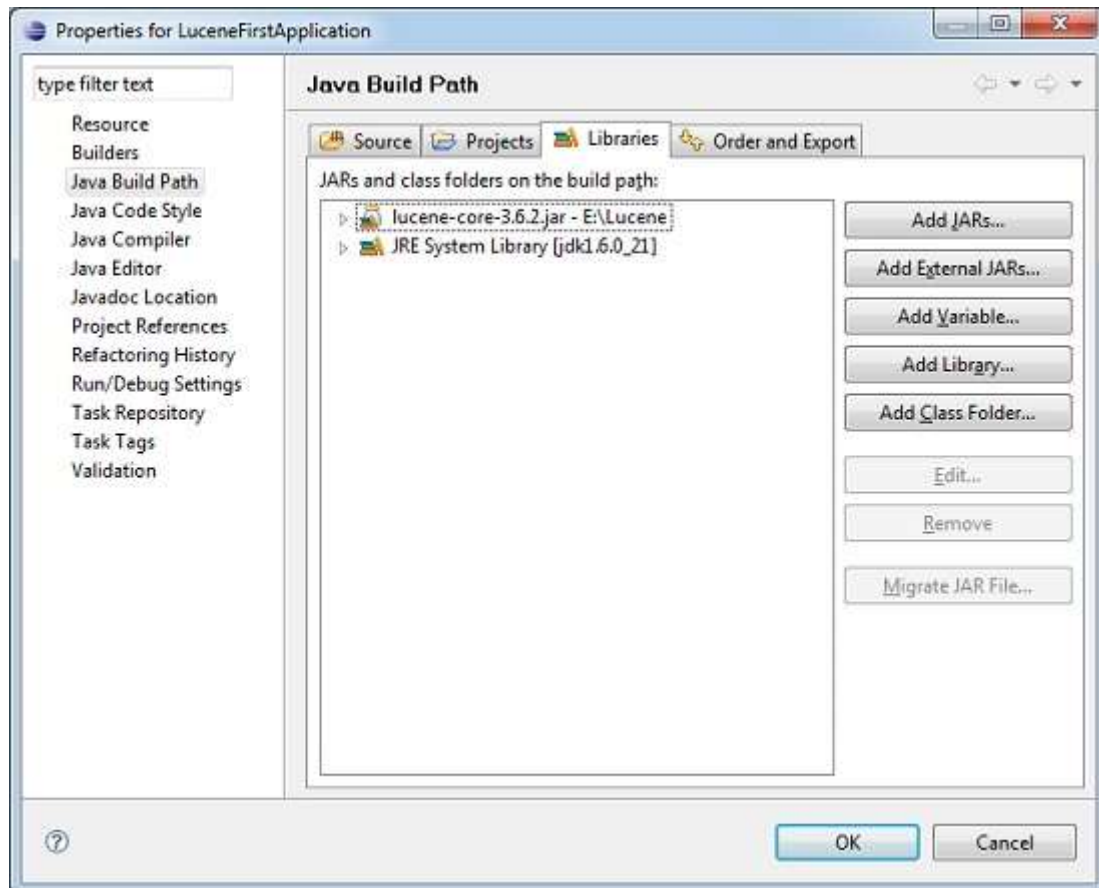


Once your project is created successfully, you will have following content in your **Project Explorer**:



Step 2: Add Required Libraries

Let us now 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 the Lucene installation directory:

- lucene-core-3.6.2

Step 3: Create Source Files

Let us now 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 the Lucene library.

```
package com.tutorialspoint.lucene;  
  
import java.io.File;  
import java.io.FileFilter;  
import java.io.FileReader;  
import java.io.IOException;  
  
import org.apache.lucene.analysis.standard.StandardAnalyzer;  
import org.apache.lucene.document.Document;  
import org.apache.lucene.document.Field;  
import org.apache.lucene.index.CorruptIndexException;  
import org.apache.lucene.index.IndexWriter;
```

```
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

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

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

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

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

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



```

        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

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

    return document;
}

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

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

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

```

Searcher.java

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

```
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 the 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

We have used 10 text files from record1.txt to record10.txt containing 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 the creation of the source, the raw data, the data directory and the index directory, you are ready for compiling and running of your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If the application runs successfully, it 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

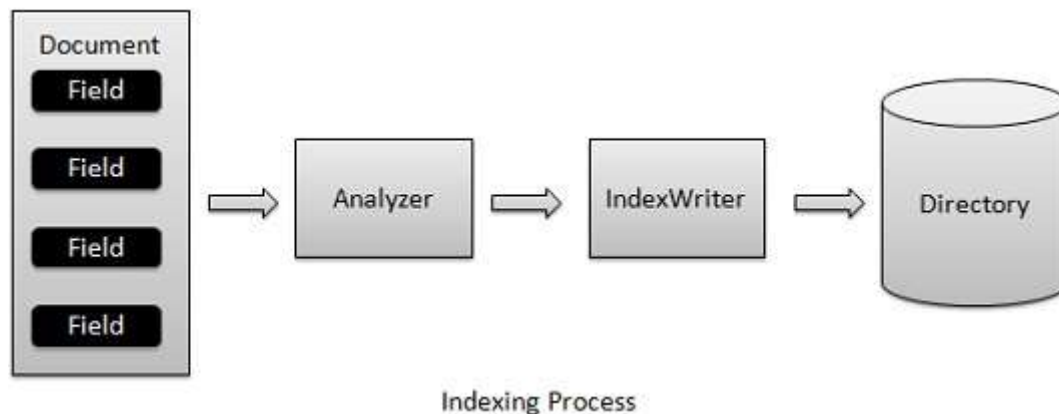
```

Once you've run the program successfully, you will have the 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	FNМ 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

4. Lucene – Indexing Classes

Indexing process is one of the core functionalities provided by Lucene. The following diagram illustrates the indexing process and the use of classes. **IndexWriter** is the most important and the core component of the indexing process.



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

Indexing Classes

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

S. No.	Class & Description
1	IndexWriter This class acts as a core component which creates/updates indexes during the indexing process.
2	Directory This class represents the storage location of the indexes.
3	Analyzer This class is responsible to analyze a document and get the tokens/words from the text which is to be indexed. Without analysis done, IndexWriter cannot create index.
4	Document

	This class represents a virtual document with Fields where the Field is an object which can contain the physical document's contents, its meta data and so on. The Analyzer can understand a Document only.
5	Field This 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. Let us assume 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 content only.

IndexWriter

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 the **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 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

Following table shows the class constructors for `IndexWriter`:

S. NO.	Constructor & Description
1	<code>IndexWriter(Directory d, Analyzer a, boolean create, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl)</code> Deprecated. Use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
2	<code>IndexWriter(Directory d, Analyzer a, boolean create, IndexWriter.MaxFieldLength mfl)</code> Deprecated. Use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
3	<code>IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl)</code> Deprecated. Use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
4	<code>IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl, IndexCommit commit)</code> Deprecated. Use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
5	<code>IndexWriter(Directory d, Analyzer a, IndexWriter.MaxFieldLength mfl)</code> Deprecated. Use <code>IndexWriter(Directory, IndexWriterConfig)</code> instead.
6	<code>IndexWriter(Directory d, IndexWriterConfig conf)</code> Constructs a new <code>IndexWriter</code> per the settings given in <code>conf</code> .

Class Methods

The following table shows the different class methods:

S. NO.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	void addDocument(Document doc, Analyzer analyzer) Adds a document to this index, using the provided analyzer instead of the value of 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, Analyzer analyzer) Atomically adds a block of documents, analyzed using the provided analyzer, with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
5	void addIndexes(Directory... dirs) Adds all segments from an array of indexes into this index.
6	void addIndexes(IndexReader... readers) Merges the provided indexes into this index.
7	void addIndexesNoOptimize(Directory... dirs) Deprecated. Use addIndexes(Directory...) instead.
8	void close() Commits all changes to an index and closes all associated files.
9	void close(boolean waitForMerges) Closes the index with or without waiting for currently running merges to finish.
10	void commit()

	Commits all pending changes (added & deleted documents, segment merges, added indexes, etc.) to the index, and syncs all referenced index files, such that a reader will see the changes and the index updates will survive an OS or machine crash or power loss.
11	void commit(Map<String,String> commitUserData) Commits all changes to the index, specifying a commitUserData Map (String -> String).
12	void deleteAll() Deletes all documents in the index.
13	void deleteDocuments(Query... queries) Deletes the document(s) matching any of the provided queries.
14	void deleteDocuments(Query query) Deletes the document(s) matching the provided query.
15	void deleteDocuments(Term... terms) Deletes the document(s) containing any of the terms.
16	void deleteDocuments(Term term) Deletes the document(s) containing term.
17	void deleteUnusedFiles() Expert: remove the index files that are no longer used.
18	protected void doAfterFlush() A hook for extending classes to execute operations after pending added and deleted documents have been flushed to the Directory but before the change is committed (new segments_N file written).
19	protected void doBeforeFlush() A hook for extending classes to execute operations before pending added and deleted documents are flushed to the Directory.
20	protected void ensureOpen()

21	protected void ensureOpen(boolean includePendingClose) Used internally to throw an AlreadyClosedException if this IndexWriter has been closed.
22	void expungeDeletes() Deprecated.
23	void expungeDeletes(boolean doWait) Deprecated.
24	protected void flush(boolean triggerMerge, boolean applyAllDeletes) Flushes all in-memory buffered updates (adds and deletes) to the Directory.
25	protected void flush(boolean triggerMerge, boolean flushDocStores, boolean flushDeletes) NOTE: flushDocStores is ignored now (hardwired to true); this method is only here for backwards compatibility.
26	void forceMerge(int maxNumSegments) This is a force merging policy to merge segments until there's <= maxNumSegments.
27	void forceMerge(int maxNumSegments, boolean doWait) Just like forceMerge(int), except you can specify whether the call should block until all merging completes.
28	void forceMergeDeletes() Forces merging of all segments that have deleted documents.
29	void forceMergeDeletes(boolean doWait) Just like forceMergeDeletes(), except you can specify whether the call should be blocked 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. Use IndexReader.open(IndexWriter,boolean) instead.
49	IndexReader getReader(int termInfosIndexDivisor) Deprecated. Use IndexReader.open(IndexWriter,boolean) instead. Furthermore, this method cannot guarantee the reader (and its sub-readers) will be opened with the termInfosIndexDivisor setting because some of them may already have been opened according to IndexWriterConfig.setReaderTermsIndexDivisor(int). You should set the requested termInfosIndexDivisor through IndexWriterConfig.setReaderTermsIndexDivisor(int) and use getReader().
50	int getReaderTermsIndexDivisor() Deprecated. Use IndexWriterConfig.getReaderTermsIndexDivisor() instead.
51	Similarity getSimilarity() Deprecated. Use IndexWriterConfig.getSimilarity() instead.
52	int getTermIndexInterval() Deprecated. Use IndexWriterConfig.getTermIndexInterval().

53	boolean getUseCompoundFile() Deprecated. Use LogMergePolicy.getUseCompoundFile().
54	long getWriteLockTimeout() Deprecated. Use IndexWriterConfig.getWriteLockTimeout()
55	boolean hasDeletions()
56	static boolean isLocked(Directory directory) Returns true if the index in the named directory is currently locked.
57	int maxDoc() Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), not counting deletions.
58	void maybeMerge() Expert: Asks the mergePolicy whether any merges are necessary now and if so, runs the requested merges and then iterate (test again if merges are needed) until no more merges are returned by the mergePolicy.
59	void merge(MergePolicy.OneMerge merge) Merges the indicated segments, replacing them in the stack with a single segment.
60	void message(String message) Prints a message to the infoStream (if non-null), prefixed with the identifying information for this writer and the thread that's calling it.
61	int numDeletedDocs(SegmentInfo info) Obtains the number of deleted docs for a pooled reader.
62	int numDocs() Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), and including deletions.
63	int numRamDocs() Expert: Returns the number of documents currently buffered in RAM.

64	void optimize() Deprecated.
65	void optimize(boolean doWait) Deprecated.
66	void optimize(int maxNumSegments) Deprecated.
67	void prepareCommit() Expert: Prepare for commit.
68	void prepareCommit(Map<String,String> commitUserData) Expert: Prepare for commit, specifying commitUserData Map (String -> String).
69	long ramSizeInBytes() Expert: Return the total size of all index files currently cached in memory.
70	void rollback() Closes the IndexWriter without committing any changes that have occurred since the last commit (or since it was opened, if commit hasn't been called).
71	String segString()
72	String segString(Iterable<SegmentInfo> infos)
73	String segString(SegmentInfo info)
74	static void setDefaultInfoStream(PrintStream infoStream) If non-null, this will be the default infoStream used by a newly instantiated IndexWriter.
75	static void setDefaultWriteLockTimeout(long writeLockTimeout) Deprecated. Use IndexWriterConfig.setDefaultWriteLockTimeout(long) instead.

76	void setInfoStream(PrintStream infoStream) If non-null, information about merges, deletes and a message when maxFieldLength is reached will be printed to this.
77	void setMaxBufferedDeleteTerms(int maxBufferedDeleteTerms) Deprecated. Use IndexWriterConfig.setMaxBufferedDeleteTerms(int) instead.
78	void setMaxBufferedDocs(int maxBufferedDocs) Deprecated. Use IndexWriterConfig.setMaxBufferedDocs(int) instead.
79	void setMaxFieldLength(int maxFieldLength) Deprecated. Use LimitTokenCountAnalyzer instead. Observe the change in the behavior - the analyzer limits the number of tokens per token stream created, while this setting limits the total number of tokens to index. This matters only if you index many multi-valued fields though.
80	void setMaxMergeDocs(int maxMergeDocs) Deprecated. Use LogMergePolicy.setMaxMergeDocs(int) directly.
81	void setMergedSegmentWarmer(IndexWriter.IndexReaderWarmer warmer) Deprecated. Use IndexWriterConfig.setMergedSegmentWarmer(org.apache.lucene.index.IndexWriter.IndexReaderWarmer) instead.
82	void setMergeFactor(int mergeFactor) Deprecated. Use LogMergePolicy.setMergeFactor(int) directly.
83	void setMergePolicy(MergePolicy mp) Deprecated. Use IndexWriterConfig.setMergePolicy(MergePolicy) instead.
84	void setMergeScheduler(MergeScheduler mergeScheduler) Deprecated. Use IndexWriterConfig.setMergeScheduler(MergeScheduler) instead
85	void setPayloadProcessorProvider(PayloadProcessorProvider pcg) Sets the PayloadProcessorProvider to use when merging payloads.

86	void setRAMBufferSizeMB(double mb) Deprecated. Use IndexWriterConfig.setRAMBufferSizeMB(double) instead.
87	void setReaderTermsIndexDivisor(int divisor) Deprecated. Use IndexWriterConfig.setReaderTermsIndexDivisor(int) instead.
88	void setSimilarity(Similarity similarity) Deprecated. Use IndexWriterConfig.setSimilarity(Similarity) instead.
89	void setTermIndexInterval(int interval) Deprecated. Use IndexWriterConfig.setTermIndexInterval(int).
90	void setUseCompoundFile(boolean value) Deprecated. Use LogMergePolicy.setUseCompoundFile(boolean).
91	void setWriteLockTimeout(long writeLockTimeout) Deprecated. Use IndexWriterConfig.setWriteLockTimeout(long) instead.
92	static void unlock(Directory directory) Forcibly unlocks the index in the named directory.
93	void updateDocument(Term term, Document doc) Updates a document by first deleting the document(s) containing term and then adding the new document.
94	void updateDocument(Term term, Document doc, Analyzer analyzer) Updates a document by first deleting the document(s) containing term and then adding the new document.
95	void updateDocuments(Term delTerm, Collection<Document> docs) Atomically deletes documents matching the provided delTerm and adds a block of documents with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
96	void updateDocuments(Term delTerm, Collection<Document> docs, Analyzer analyzer)

	Atomically deletes documents matching the provided delTerm and adds a block of documents, analyzed using the provided analyzer, with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
97	boolean verbose() Returns true if verbosing is enabled (i.e., infoStream)
98	void waitForMerges() Waits for any currently outstanding merges to finish.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

Directory

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

Class Declaration

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

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

Field

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

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

Class Constructors

The following table shows a Class Constructor:

S. NO.	Constructor & Description
1	Directory()

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void clearLock(String name) Attempt to clear (forcefully unlock and remove) the specified lock.
2	abstract void close() Closes the store.
3	static void copy(Directory src, Directory dest, boolean closeDirSrc) Deprecated. Should be replaced with calls to copy (Directory, String, String) for every file that needs copying. You can use the following code: <pre> IndexFileNameFilter filter = IndexFileNameFilter.getFilter(); for (String file : src.listAll()) { if (filter.accept(null, file)) { src.copy(dest, file, file); } } </pre>
4	void copy(Directory to, String src, String dest) Copies the file src to Directory under the new file name dest .
5	abstract IndexOutput createOutput(String name) Creates a new, empty file in the directory with the given name.
6	abstract void deleteFile(String name)

	Removes an existing file in the directory.
7	protected void ensureOpen() =
8	abstract boolean fileExists(String name) Returns true if a file with the given name exists.
9	abstract long fileLength(String name) Returns the length of a file in the directory.
10	abstract long fileModified(String name) Deprecated.
11	LockFactory getLockFactory() Gets the LockFactory that this Directory instance is using for its locking implementation.
12	String getLockID() Returns a string identifier that uniquely differentiates this Directory instance from other Directory instances.
13	abstract String[] listAll() Returns an array of strings, one for each file in the directory.
14	Lock makeLock(String name) Constructs a Lock.
15	abstract IndexInput openInput(String name) Returns a stream reading an existing file.
16	IndexInput openInput(String name, int bufferSize) Returns a stream reading an existing file, with the specified read buffer size.
17	void setLockFactory(LockFactory lockFactory) Sets the LockFactory that this Directory instance should use for its locking implementation.

18	void sync(Collection<String> names) Ensures that any rights to these files are moved to stable storage.
19	void sync(String name) Deprecated. Use sync(Collection) instead. For easy migration you can change your code to call sync(Collections.singleton(name))
20	String toString()
21	abstract void touchFile(String name) Deprecated. Lucene never uses this API; it will be removed in 4.0.

Methods Inherited

This class inherits methods from the following classes:

java.lang.Object

Analyzer

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

The following table shows a class constructor:

S. NO.	Constructor & Description
1	protected Analyzer()

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void close() Frees persistent resources used by this Analyzer.
2	int getOffsetGap(Fieldable field) Just like getPositionIncrementGap(java.lang.String), except for Token offsets instead.
3	int getPositionIncrementGap(String fieldName) Invoked before indexing a Fieldable instance if terms have already been added to that field.
4	protected Object getPreviousTokenStream() Used by Analyzers that implement reusableTokenStream to retrieve previously saved TokenStreams for re-use by the same thread.
5	TokenStream reusableTokenStream(String fieldName, Reader reader) Creates a TokenStream that is allowed to be re-used from the previous time that the same thread called this method.
6	protected void setPreviousTokenStream(Object obj) Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the same thread.
7	abstract TokenStream tokenStream(String fieldName, Reader reader) Creates a TokenStream which tokenizes all the text in the provided Reader.

Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

Document

Document represents a virtual document with Fields where Field is an 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

Following tables shows a class constructor:

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

Class Methods

Following table shows the different class methods:

S. No.	Method & Description
1	void clearLock(String name) Attempt to clear (forcefully unlock and remove) the specified lock.
2	void add(Fieldable field) Adds a field to a document.
3	String get(String name) Returns the string value of the field with the given name if any in this document, or null.
4	byte[] getBinaryValue(String name)

	Returns an array of bytes for the first (or only) field that has the name specified as the method parameter.
5	byte[][] getBinaryValues(String name) Returns an array of byte arrays for the fields that have the name specified as the method parameter.
6	float getBoost() Returns, at indexing time, the boost factor as set by setBoost(float).
7	Field getField(String name) Deprecated. Use getFieldable(java.lang.String) instead and cast depending on data type.
8	Fieldable getFieldable(String name) Returns a field with the given name if any exist in this document, or null.
9	Fieldable[] getFieldables(String name) Returns an array of Fieldables with the given name.
10	List<Fieldable> getFields() Returns a List of all the fields in a document.
11	Field[] getFields(String name) Deprecated. Use getFieldable(java.lang.String) instead and cast depending on data type.
12	String[] getValues(String name) Returns an array of values of the field specified as the method parameter.
13	void removeField(String name) Removes field with the specified name from the document.
14	void removeFields(String name) Removes all fields with the given name from the document.

15	void setBoost(float boost) Sets a boost factor for hits on any field of this document.
16	String toString() Prints the fields of a document for human consumption.

Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

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. 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

Following table shows a list of class constructors:

S. No.	Constructor & Description
1	Field(String name, boolean internName, String value, Field.Store store, Field.Index index, Field.TermVector termVector) Creates a field by specifying its name, value and how it will be saved in the index.
2	Field(String name, byte[] value)

	Creates a stored field with binary value.
3	Field(String name, byte[] value, Field.Store store) Deprecated.
4	Field(String name, byte[] value, int offset, int length) Creates a stored field with binary value.
5	Field(String name, byte[] value, int offset, int length, Field.Store store) Deprecated.
6	Field(String name, Reader reader) Creates a tokenized and indexed field that is not stored.
7	Field(String name, Reader reader, Field.TermVector termVector) Creates a tokenized and indexed field that is not stored, optionally with storing term vectors.
8	Field(String name, String value, Field.Store store, Field.Index index) Creates a field by specifying its name, value and how it will be saved in the index.
9	Field(String name, String value, Field.Store store, Field.Index index, Field.TermVector termVector) Creates a field by specifying its name, value and how it will be saved in the index.
10	Field(String name, TokenStream tokenStream) Creates a tokenized and indexed field that is not stored.
11	Field(String name, TokenStream tokenStream, Field.TermVector termVector) Creates a tokenized and indexed field that is not stored, optionally with storing term vectors.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void clearLock(String name) Attempts to clear (forcefully unlock and remove) the specified lock.
2	Reader readerValue() The value of the field as a Reader, or null.
3	void setTokenStream(TokenStream tokenStream) Expert: sets the token stream to be used for indexing and causes isIndexed() and isTokenized() to return true.
4	void setValue(byte[] value) Expert: changes the value of this field.
5	void setValue(byte[] value, int offset, int length) Expert: changes the value of this field.
6	void setValue(Reader value) Expert: changes the value of this field.
7	void setValue(String value) Expert: changes the value of this field.
8	String stringValue() The value of the field as a String, or null.
9	TokenStream tokenStreamValue() The TokenStream for this field to be used when indexing, or null.

Methods Inherited

This class inherits methods from the following classes:

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

5. Lucene – Searching Classes

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

Searching Classes

Following is a list of commonly-used classes during searching process.

S. No.	Class & Description
1	IndexSearcher This class act as a core component which reads/searches indexes created after the 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 a simple container of pointers to point to documents which are the output of a search result.

IndexSearcher

This class acts as a core component which reads/searches indexes during the 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

The following table shows a list of class constructors:

S. No.	Constructor & Description
1	IndexSearcher(Directory path) Deprecated. Use IndexSearcher(IndexReader) instead.
2	IndexSearcher(Directory path, boolean readOnly) Deprecated. Use IndexSearcher(IndexReader) instead.
3	IndexSearcher(IndexReader r) Creates a searcher searching the provided index.
4	IndexSearcher(IndexReader r, ExecutorService executor) Runs searches for each segment separately, using the provided ExecutorService.
5	IndexSearcher(IndexReader reader, IndexReader[] subReaders, int[] docStarts) Expert: directly specify the reader, subReaders and their docID starts.
6	IndexSearcher(IndexReader reader, IndexReader[] subReaders, int[] docStarts, ExecutorService executor)

	Expert: directly specify the reader, subReaders and their docID starts, and an ExecutorService.
--	---

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void close() Note that the underlying IndexReader is not closed, if IndexSearcher was constructed with IndexSearcher(IndexReader).
2	Weight createNormalizedWeight(Query query) Creates a normalized weight for a top-level Query.
3	Document doc(int docID) Returns the stored fields of document ID.
4	Document doc(int docID, FieldSelector fieldSelector) Get the Document at the nth position.
5	int docFreq(Term term) Returns total docFreq for this term.
6	Explanation explain(Query query, int doc) Returns an Explanation that describes how a doc scored against a query.
7	Explanation explain(Weight weight, int doc) Expert: Low-level implementation method that returns an Explanation that describes how a doc scored against weight.
8	protected void gatherSubReaders(List allSubReaders, IndexReader r)
9	IndexReader getIndexReader() Returns the IndexReader this searches.

10	Similarity getSimilarity() Expert: Returns the Similarity implementation used by this Searcher.
11	IndexReader[] getSubReaders() Returns the atomic subReaders used by this searcher.
12	int maxDoc() Expert: Returns one greater than the largest possible document number.
13	Query rewrite(Query original) Expert: Called to re-write queries into primitive queries.
14	void search(Query query, Collector results) Lower-level search API.
15	void search(Query query, Filter filter, Collector results) Lower-level search API.
16	TopDocs search(Query query, Filter filter, int n) Finds the top n hits for query, applying filter if non-null.
17	TopFieldDocs search(Query query, Filter filter, int n, Sort sort) Search implementation with arbitrary sorting.
18	TopDocs search(Query query, int n) Finds the top n hits for query.
19	TopFieldDocs search(Query query, int n, Sort sort) Search implementation with arbitrary sorting and no filter.
20	void search(Weight weight, Filter filter, Collector collector) Lower-level search API.
21	TopDocs search(Weight weight, Filter filter, int nDocs) Expert: Low-level search implementation.

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

Methods Inherited

This class inherits methods from the following classes:

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

Term

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

The following table shows a list of class constructors:

S. No.	Constructor & Description
1	Term(String fld) Constructs a Term with the given field and empty text.
2	Term(String fld, String txt) Constructs a Term with the given field and text.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	int compareTo(Term other) Compares two terms, returning a negative integer if this term belongs before the argument, zero if this term is equal to the argument, and a positive integer if this term belongs after the argument.
3	Term createTerm(String text) Optimized construction of new Terms by reusing same field as this Term - avoids field.intern() overhead.

4	boolean equals(Object obj)
5	String field() Returns the field of this term, an interned string.
6	int hashCode()
7	String text() Returns the text of this term.
8	String toString()

Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

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.

Class Declaration

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

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

Class Constructors

The following table shows a class constructor:

S. No.	Constructor & Description
1	Query()

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	Object clone() Returns a clone of this query.
2	Query combine(Query[] queries) Expert: Called when re-writing queries under MultiSearcher.
3	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
4	boolean equals(Object obj)
5	void extractTerms(Set<Term> terms) Expert: Adds all terms occurring in this query to the terms set.
6	float getBoost() Gets the boost for this clause.
7	Similarity getSimilarity(Searcher searcher) Deprecated. Instead of using "runtime" subclassing/delegation, subclass the Weight instead.
8	int hashCode()
9	static Query mergeBooleanQueries(BooleanQuery... queries) Expert: Merges the clauses of a set of BooleanQuery's into a single BooleanQuery.
10	Query rewrite(IndexReader reader) Expert: Called to re-write queries into primitive queries.
11	void setBoost(float b) Sets the boost for this query clause to b.

12	String toString() Prints a query to a string.
13	abstract String toString(String field) Prints a query to a string, with field assumed to be the default field and omitted.
14	Weight weight(Searcher searcher) Deprecated. Never ever use this method in Weight implementations. Subclasses of Query should use createWeight(org.apache.lucene.search.Searcher), instead.

Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

TermQuery

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

The following table shows a class constructor:

S. No.	Constructor & Description
1	TermQuery(Term t) Constructs a query for the term t.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
3	boolean equals(Object o) Returns true iff o is equal to this.
4	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.
5	Term getTerm() Returns the term of this query.
6	int hashCode() Returns a hash code value for this object.
7	String toString(String field) Prints a user-readable version of this query.

Methods Inherited

This class inherits methods from the following classes:

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

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.

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

The following table shows a class constructor:

S. NO.	Constructor & Description
1	TopDocs(int totalHits, ScoreDoc[] scoreDocs, float maxScore)

Class Methods

The following table shows the different class methods:

S. NO.	Method & Description
1	getMaxScore() Returns the maximum score value encountered.
2	static TopDocs merge(Sort sort, int topN, TopDocs[] shardHits) Returns a new TopDocs, containing topN results across the provided TopDocs, sorting by the specified Sort.
3	void setMaxScore(float maxScore) Sets the maximum score value encountered.

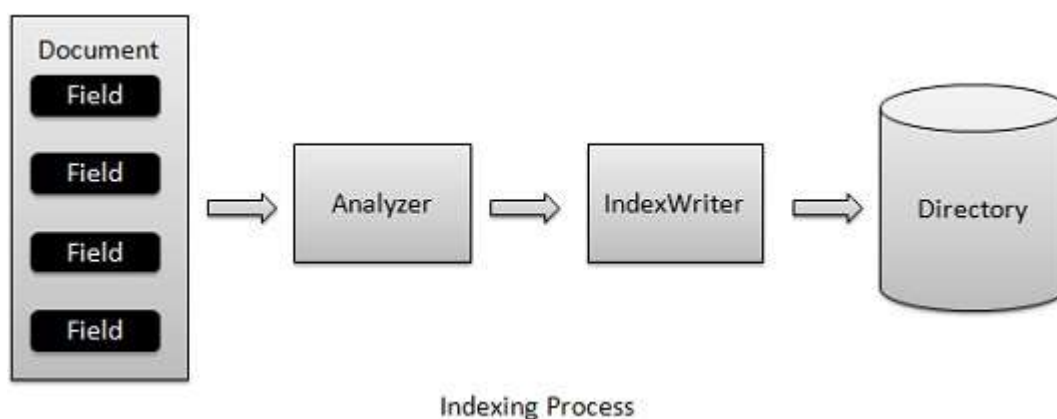
Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

6. Lucene – Indexing Process

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



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

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

Create a document

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

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

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

```

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

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

return document;
}

```

Create a IndexWriter

IndexWriter class acts as a core component which creates/updates indexes during indexing process. Follow these steps to create a IndexWriter:

Step 1: Create object of IndexWriter.

Step 2: Create a Lucene directory which should point to location where indexes are to be stored.

Step 3: 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

The following program shows how to start an indexing process:

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

Example Application

To test the indexing process, we need to create a Lucene application test.

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

```
package com.tutorialspoint.lucene;

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

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

```
public class Indexer {

    private IndexWriter writer;

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

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

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

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

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

        document.add(contentField);
```

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

        return document;
    }

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

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

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

LuceneTester.java

This class is used to test the indexing capability of the 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

We have used 10 text files from record1.txt to record10.txt containing 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 the creation of the source, the raw data, the data directory and the index directory, you can proceed by compiling and running your program. To do this, keep the LuceneTester.Java file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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 the 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

7. 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 a list of commonly-used operations during indexing process.

S. No.	Operation & Description
1	Add Document This operation is used in the initial stage of the indexing process to create the indexes on the newly available content.
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 specify a way or control the ways in which the contents of a field are to be made searchable.

Add Document Operation

Add document is one of the core operations of the indexing process.

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

We will now show you a step-wise approach and help you understand how to add a document using a basic example.

Add a document to an index

Follow these steps to add a document to an index:

Step 1: Create a method to get a Lucene document from a text file.

Step 2: Create various fields which are key value pairs containing keys as names and values as contents to be indexed.

Step 3: Set field to be analyzed or not. In our case, only the content is to be analyzed as it can contain data such as a, am, are, an etc. which are not required in search operations.

Step 4: Add the newly-created fields to the document object and return it to the caller method.

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

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

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

    return document;
}
```

Create a IndexWriter

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

Follow these steps to create a IndexWriter:

Step 1: Create object of IndexWriter.

Step 2: Create a Lucene directory which should point to location where indexes are to be stored.

Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;

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

Add Document and Start Indexing Process

Following two are the ways to add the document.

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

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

Example Application

To test the indexing process, we need to create Lucene application test.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. To understand the indexing process, you can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter.

2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep the rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

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

public class TextFileFilter implements FileFilter {

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

Indexer.java

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

```
package com.tutorialspoint.lucene;

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

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

public class Indexer {

    private IndexWriter writer;

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

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

```

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

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

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

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

    return document;
}

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

public int createIndex(String dataDirPath, FileFilter filter)
    throws IOException{
    //get all files in the data directory

```

```

        File[] files = new File(dataDirPath).listFiles();

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

```

LuceneTester.java

This class is used to test the indexing capability of the 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

We have used 10 text files from record1.txt to record10.txt containing 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 the creation of the 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 your application runs successfully, it 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	FNМ 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

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

We will now show you a step-wise approach and help you understand how to update document using a basic example.

Update a Document to an Index

Follow this step to update a document to an index:

Step 1: 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 an IndexWriter

Follow these steps to create an IndexWriter:

Step 1: IndexWriter class acts as a core component which creates/updates indexes during the indexing process.

Step 2: Create object of IndexWriter.

Step 3: Create a Lucene directory which should point to location where indexes are to be stored.

Step 4: Initialize the IndexWriter object created with the index directory, a standard analyzer having version information and other required/optional parameters.

```
private IndexWriter writer;

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

Update document and start reindexing process

Following are the two ways to update the document.

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

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

Example Application

To test the indexing process, let us create a Lucene application test.

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

TextFileFilter.java

This class is used as a **.txt** file filter.

```
package com.tutorialspoint.lucene;

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

```

public class TextFileFilter implements FileFilter {

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

```

Indexer.java

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

Here, we have used 10 text files from record1.txt to record10.txt containing 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 the creation of the source, the raw data, the data directory and the index directory, you can proceed with the compiling and running of your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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 above program successfully, you will have the 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

Delete Document Operation

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

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

We will now show you a step-wise approach and make you understand how to delete a document using a basic example.

Delete a document from an index

Follow these steps to delete a document from an index:

Step 1: 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 an IndexWriter

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

Follow these steps to create an IndexWriter:

Step 1: Create object of IndexWriter.

Step 2: Create a Lucene directory which should point to a location where indexes are to be stored.

Step 3: Initialize the IndexWriter object created with the index directory, a standard analyzer having the 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 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

To test the indexing process, let us create a Lucene application test.

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

LuceneConstants.java

This class provides various constants that can 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 thereby, making it searchable using the 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 the 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

We've used 10 text files from record1.txt to record10.txt containing 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 the creation of the source, the raw data, the data directory and the index directory, you can compile and run your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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	FNМ 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

Field Options

Field is the most important unit of the 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 the Field Options which state how much a field is to be searchable.

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

We will now show you a step-wise approach and help you understand the various Field Options using a basic example.

Various Field Options

Following are the various field options:

- **Index.ANALYZED** — In this, we first analyze, then do indexing. This is used for normal text indexing. Analyzer will break the field's value into stream of tokens and each token is searchable separately.
- **Index.NOT_ANALYZED** — In this, we do not analyze but do indexing. This is used for complete text indexing. For example, person's names, URL etc.
- **Index.ANALYZED_NO_NORMS** — This is a variant of **Index.ANALYZED**. The Analyzer will break the field's value into stream of tokens and each token is searchable separately. However, the NORMS are not stored in the indexes. NORMS are used to boost searching and this often ends up consuming a lot of memory.
- **Index.Index.NOT_ANALYZED_NO_NORMS** — This is variant of **Index.NOT_ANALYZED**. Indexing is done but NORMS are not stored in the indexes.
- **Index.NO** — Field value is not searchable.

Use of Field Options

Following are the different ways in which the Field Options can be used:

- To create a method to get a Lucene document from a text file.
- To create various types of fields which are key value pairs containing keys as names and values as contents to be indexed.
- To set field to be analyzed or not. In our case, only content is to be analyzed as it can contain data such as a, am, are, an, etc. which are not required in search operations.
- To 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.CONTENT,
        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

To test the indexing process, we need to create a Lucene application test.

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

```

package com.tutorialspoint.lucene;

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

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

public class Indexer {

    private IndexWriter writer;

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

```

```

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

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

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

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

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

    return document;
}

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

```

```

        writer.addDocument(document);
    }

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

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

```

LuceneTester.java

This class is used to test the indexing capability of the 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

We have used 10 text files from record1.txt to record10.txt containing 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 the creation of the source, the raw data, the data directory and the index directory, you can compile and run your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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

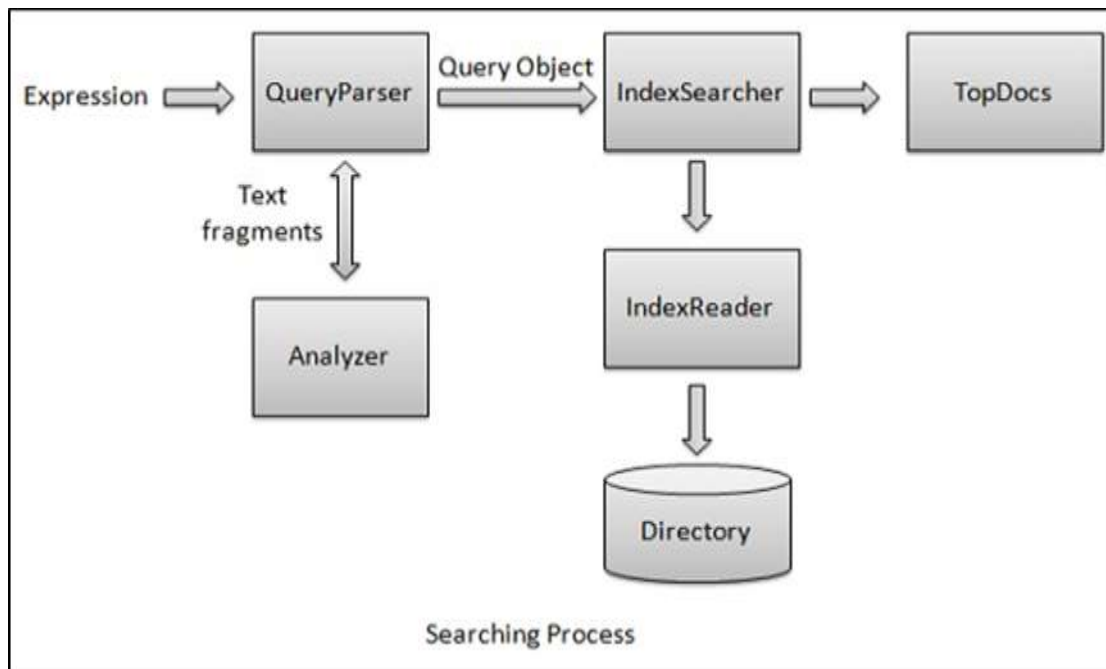
```

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

8. Lucene – Search Operation

The process of searching is one of the core functionalities provided by Lucene. Following diagram illustrates the process and its use. *IndexSearcher* is one of the core components of the 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.

We will now show you a step-wise approach and help you understand the indexing process using a basic example.

Create a QueryParser

QueryParser class parses the user entered input into Lucene understandable format query. Follow these steps to create a *QueryParser*:

Step 1: Create object of *QueryParser*.

Step 2: Initialize the *QueryParser* object created with a standard analyzer having version information and index name on which this query is to be 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. Follow these steps to create a IndexSearcher:

Step 1: Create object of IndexSearcher.

Step 2: Create a Lucene directory which should point to location where indexes are to be stored.

Step 3: 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

Follow these steps to make search:

Step 1: Create a Query object by parsing the search expression through QueryParser.

Step 2: Make search by calling the 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

The following program shows how to get the document.

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

Close IndexSearcher

The following program shows how to close the IndexSearcher.

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

Example Application

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

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

LuceneConstants.java

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

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

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

TextFileFilter.java

This class is used as a **.txt** file filter.

```
package com.tutorialspoint.lucene;  
  
import java.io.File;  
import java.io.FileFilter;  
  
public class TextFileFilter implements FileFilter {  
  
    @Override  
    public boolean accept(File pathname) {  
        return pathname.getName().toLowerCase().endsWith(".txt");  
    }  
}
```

Searcher.java

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

We have used 10 text files named record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTesterapplication**. If your application runs successfully, it will print the following message in Eclipse IDE's console:

```

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

```

9. Lucene – Query Programming

We have seen in previous chapter **Lucene - Search Operation**, Lucene uses IndexSearcher to make searches and it uses the Query object created by QueryParser as the input. In this chapter, we are going to discuss various types of Query objects and the different 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 the indexing process.
2	TermRangeQuery TermRangeQuery is 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 the edit distance algorithm.
8	MatchAllDocsQuery MatchAllDocsQuery as the name suggests matches all the documents.

TermQuery

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

Class Declaration

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

```
public class TermQuery
    extends Query
```

Class Constructors

The following table shows a class constructor:

S. No.	Constructor & Description
1	TermQuery(Term t) Constructs a query for the term t.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void addDocument(Document doc) Adds a document to this index.
2	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.

3	boolean equals(Object o) Returns true if object o is equal to this.
4	void extractTerms(Set<Term> terms) Expert: Adds all terms occurring in this query to the terms set.
5	Term getTerm() Returns the term of this query.
6	int hashCode() Returns a hash code value for this object.
7	String toString(String field) Prints a user-readable version of this query.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

    System.out.println(hits.totalHits +
        " documents found. Time :" + (endTime - startTime) + "ms");
```



```

    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: "+ doc.get(LuceneConstants.FILE_PATH));
    }
    searcher.close();
}

```

Example Application

To test search using TermQuery, let us create a test Lucene application.

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

LuceneConstants.java

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

```

package com.tutorialspoint.lucene;

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

```

Searcher.java

This class is used to read the indexes made on raw data and searches data using the 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 the 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 from record1.txt to record10.txt containing 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 in the chapter *Lucene - Indexing Process*, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

```

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

```

TermRangeQuery

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

Class Declaration

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

```

public class TermRangeQuery
    extends MultiTermQuery

```

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	TermRangeQuery(String field, String lowerTerm, String upperTerm, boolean includeLower, boolean includeUpper) Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.
2	TermRangeQuery(String field, String lowerTerm, String upperTerm, boolean includeLower, boolean includeUpper, Collator collator) Constructs a query selecting all terms greater/equal than lowerTerm but less/equal than upperTerm.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	boolean equals(Object obj)
2	Collator getCollator() Returns the collator used to determine range inclusion, if any.
3	protected FilteredTermEnum getEnum(IndexReader reader) Construct the enumeration to be used, expanding the pattern term.
4	String getField() Returns the field name for this query.
5	String getLowerTerm() Returns the lower value of this range query.
6	String getUpperTerm() Returns the upper value of this range query.
7	int hashCode()

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

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

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

```
}
```

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

Searcher.java

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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter Lucene - Indexing Process, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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

```

PrefixQuery

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

Class Declaration

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

```

public class PrefixQuery
    extends MultiTermQuery

```

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
--------	---------------------------

1	PrefixQuery(Term prefix) Constructs a query for the term starting with prefix.
---	--

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	boolean equals(Object o) Returns true if object o is equal to this.
2	protected FilteredTermEnum getEnum(IndexReader reader) Constructs the enumeration to be used, expanding the pattern term.
3	Term getPrefix() Returns the prefix of this query.
4	int hashCode() Returns a hash code value for this object.
5	String toString(String field) Prints a user-readable version of this query.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

```
private void searchUsingPrefixQuery(String searchQuery)
    throws IOException, ParseException{
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    //create a term to search file name
```

```

Term term = new Term(LuceneConstants.FILE_NAME, searchQuery);
//create the term query object
Query query = new PrefixQuery(term);
//do the search
TopDocs hits = searcher.search(query);
long endTime = System.currentTimeMillis();

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

```

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

Searcher.java

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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the LuceneTester.Java file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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

```

BooleanQuery

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

Class Declaration

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

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

Fields

Following is the field for the BooleanQuery:

- protected int minNrShouldMatch

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	BooleanQuery() Constructs an empty Boolean query.
2	BooleanQuery(boolean disableCoord) Constructs an empty Boolean query.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void add(BooleanClause clause) Adds a clause to a Boolean query.
2	void add(Query query, BooleanClause.Occur occur) Adds a clause to a boolean query.
3	List<BooleanClause> clauses() Returns the list of clauses in this query.
4	Object clone() Returns a clone of this query.
5	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
6	boolean equals(Object o) Returns true if object 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() Returns 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 if Similarity.coord(int,int) is disabled in scoring for this query instance.
13	Iterator<BooleanClause> iterator() Returns an iterator on the clauses in this query.
14	Query rewrite(IndexReader reader) Expert: Called to re-write queries into primitive queries.
15	static void setMaxClauseCount(int maxClauseCount) Sets the maximum number of clauses permitted per BooleanQuery.
16	void setMinimumNumberShouldMatch(int min) Specifies a minimum number of the optional BooleanClauses which must be satisfied.
17	String toString(String field) Prints a user-readable version of this query.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

```

//create the term query object
Query query1 = new TermQuery(term1);

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

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

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

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

```

Example Application

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

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

3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure the 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 the Lucene library.

```
package com.tutorialspoint.lucene;

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

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

```
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

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

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

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

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

    public void close() throws IOException{
```



```
        indexSearcher.close();  
    }  
}
```

LuceneTester.java

This class is used to test the searching capability of the 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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the LuceneTester.java file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

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

PhraseQuery

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

Class Declaration

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

```
public class PhraseQuery
    extends Query
```

Class Constructors

The following table shows a class constructor:

S. No.	Constructor & Description
1	PhraseQuery() Constructs an empty phrase query.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void add(Term term) Adds a term to the end of the query phrase.
2	void add(Term term, int position) Adds a term to the end of the query phrase.
3	Weight createWeight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
4	boolean equals(Object o) Returns true if object o is equal to this.
5	void extractTerms(Set<Term> queryTerms) Expert: Adds all terms occurring in this query to the terms set.
6	int[] getPositions() Returns the relative positions of terms in this phrase.
7	int getSlop() Returns the slop.
8	Term[] getTerms() Returns the set of terms in this phrase.
9	int hashCode() Returns a hash code value for this object.
10	Query rewrite(IndexReader reader) Expert: Called to re-write queries into primitive queries.
11	void setSlop(int s)

	Sets the number of other words permitted between words in query phrase.
12	String toString(String f) Prints a user-readable version of this query.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

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

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

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

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

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

Searcher.java

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

```
package com.tutorialspoint.lucene;

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

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

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

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

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

```

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

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

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

```

LuceneTester.java

This class is used to test the searching capability of the 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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

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

WildcardQuery

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

Class declaration

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

```
public class WildcardQuery
    extends MultiTermQuery
```

Fields

- protected Term term

Class constructors

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

Class methods

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

Methods inherited

This class inherits methods from the following classes:

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

Usage

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

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

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

Searcher.java

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

```
package com.tutorialspoint.lucene;

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

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

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

public class Searcher {

    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

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

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

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

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

```

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

```

LuceneTester.java

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

```

package com.tutorialspoint.lucene;

import java.io.IOException;

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

public class LuceneTester {

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

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

```

```

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

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

```

Data & Index directory creation

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

Running the Program:

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

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

FuzzyQuery

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

Class Declaration

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

```
public class FuzzyQuery
    extends MultiTermQuery
```

Fields

Following are the fields for the FuzzyQuery:

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

Class constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	FuzzyQuery(Term term) Calls FuzzyQuery(term, 0.5f, 0, Integer.MAX_VALUE)
2	FuzzyQuery(Term term, float minimumSimilarity) Calls FuzzyQuery(term, minimumSimilarity, 0, Integer.MAX_VALUE)
3	FuzzyQuery(Term term, float minimumSimilarity, int prefixLength) Calls FuzzyQuery(term, minimumSimilarity, prefixLength, Integer.MAX_VALUE)

4	FuzzyQuery(Term term, float minimumSimilarity, int prefixLength, int maxExpansions) Create a new FuzzyQuery that will match terms with a similarity of at least minimum Similarity to term
---	--

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	boolean equals(Object obj)
2	protected FilteredTermEnum getEnum(IndexReader reader) Constructs the enumeration to be used, expanding the pattern term.
3	float getMinSimilarity() Returns the minimum similarity that is required for this query to match.
4	int getPrefixLength() Returns the non-fuzzy prefix length.
5	Term getTerm() Returns the pattern term.
6	int hashCode()
7	String toString(String field) Prints a query to a string, with field assumed to be the default field and omitted.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

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

Example Application

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

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

3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure the 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 the 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 the 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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep LuceneTester.Java file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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
```

MatchAllDocsQuery

MatchAllDocsQuery as the 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. NO.	Constructor & Description
1	MatchAllDocsQuery()
2	MatchAllDocsQuery(String normsField)

Class Methods

S. NO.	Method & Description
1	Weight create Weight(Searcher searcher) Expert: Constructs an appropriate Weight implementation for this query.
2	boolean equals(Object o)
3	void extractTerms(Set<Term> terms) Expert: adds all terms occurring in this query to the terms set.

4	int hashCode()
5	String to String(String field) Prints a query to a string, with field assumed to be the default field and omitted.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

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


Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

Searcher.java

This class is used to read the indexes made on raw data and searches data using the 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 the 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 from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can proceed by compiling and running your program. To do this, keep the `LuceneTester.java` file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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
```

10. Lucene – Analysis

In one of our previous chapters, we have seen that Lucene uses *IndexWriter* to analyze the *Document(s)* using the *Analyzer* and then creates/open/edit indexes as required. In this chapter, we are going to discuss the various types of *Analyzer* objects and other relevant objects which are used during the analysis process. Understanding the 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.

S. No.	Class & Description
1	Token Token represents text or word in a document with relevant details like its metadata (position, start offset, end offset, token type and its position increment).
2	TokenStream TokenStream is an output of the analysis process and it comprises of a series of tokens. It is an abstract class.
3	Analyzer This is an abstract base class for each and every type of Analyzer.
4	WhitespaceAnalyzer This analyzer splits the text in a document based on whitespace.
5	SimpleAnalyzer This analyzer splits the text in a document based on non-letter characters and puts the text in lowercase.
6	StopAnalyzer This analyzer works just as the SimpleAnalyzer and removes the common words like 'a', 'an', 'the', etc.
7	StandardAnalyzer

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

Token

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

Class Declaration

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

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

Fields

Following are the fields for the **org.apache.lucene.analysis.Token** class:

- **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

The following table shows the different class constructors:

S. No.	Constructor & Description
1	Token() Constructs a Token with null text.
2	Token(char[] startTermBuffer, int termBufferOffset, int termBufferLength, int start, int end) Constructs a Token with the given term buffer (offset & length), start and end offsets
3	Token(int start, int end) Constructs a Token with null text and start & end offsets.

4	Token(int start, int end, int flags) Constructs a Token with null text and start & end offsets plus flags.
5	Token(int start, int end, String typ) Constructs a Token with null text and start/ end offsets plus the Token type.
6	Token(String text, int start, int end) Constructs a Token with the given term text, and start/ end offsets.
7	Token(String text, int start, int end, int flags) Constructs a Token with the given text, start/ end offsets, and type.
8	Token(String text, int start, int end, String typ) Constructs a Token with the given text, start/ end offsets, and type.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void clear() Resets the term text, payload, flags, and positionIncrement, startOffset, endOffset and token type to default.
2	Object clone() This is a shallow clone.
3	Token clone(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Makes a clone, but replaces the term buffer & start/end offset in the process.
4	void copyTo(AttributeImpl target) Copies the values from this Attribute into the passed-in target attribute.
5	int endOffset()

	Returns the Token's ending offset; one greater than the position of the last character corresponding to this token in the source text.
6	boolean equals(Object obj)
7	int getFlags() Gets the bitset for any bits that have been set.
8	Payload getPayload() Returns this Token's payload.
9	int getPositionIncrement() Returns the position increment of this Token.
10	int getPositionLength() Get the position length.
11	int hashCode()
12	void reflectWith(AttributeReflector reflector) This method is for introspection of attributes, it should simply add the key/values this attribute holds to the given AttributeReflector.
13	Token reinit(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributeImpl.copyBuffer(char[], int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE
14	Token reinit(char[] newTermBuffer, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributeImpl.copyBuffer(char[], int, int), setStartOffset(int), setEndOffset(int), setType(java.lang.String)
15	Token reinit(String newTerm, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributeImpl.append(CharSequence), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE

16	Token reinit(String newTerm, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset) Shorthand for calling clear(), CharTermAttributeImpl.append(CharSequence, int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String) on Token.DEFAULT_TYPE
17	Token reinit(String newTerm, int newTermOffset, int newTermLength, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributeImpl.append(CharSequence, int, int), setStartOffset(int), setEndOffset(int) setType(java.lang.String)
18	Token reinit(String newTerm, int newStartOffset, int newEndOffset, String newType) Shorthand for calling clear(), CharTermAttributeImpl.append(CharSequence), setStartOffset(int), setEndOffset(int) setType(java.lang.String)
19	void reinit(Token prototype) Copies the prototype token's fields into this one.
20	void reinit(Token prototype, char[] newTermBuffer, int offset, int length) Copies the prototype token's fields into this one, with a different term.
21	void reinit(Token prototype, String newTerm) Copies the prototype token's fields into this one, with a different term.
22	void setEndOffset(int offset) Sets the ending offset.
23	void setFlags(int flags)
24	void setOffset(int startOffset, int endOffset) Sets the starting and ending offset.
25	void setPayload(Payload payload) Sets this Token's payload.
26	void setPositionIncrement(int positionIncrement)

	Sets the position increment.
27	void setPositionLength(int positionLength) Set the position length.
28	void setStartOffset(int offset) Set the starting offset.
29	void setType(String type) Sets 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

TokenStream

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

Class Declaration

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

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

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	protected <code>TokenStream()</code> A <code>TokenStream</code> that uses the default attribute factory.
2	protected <code>TokenStream(AttributeSource.AttributeFactory factory)</code> A <code>TokenStream</code> that uses the supplied <code>AttributeFactory</code> for creating new <code>Attribute</code> instances.
3	protected <code>TokenStream(AttributeSource input)</code> A <code>TokenStream</code> that uses the same attributes as the supplied one.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void <code>close()</code> Releases resources associated with this stream.
2	void <code>end()</code> This method is called by the consumer after the last token has been consumed, after <code>incrementToken()</code> returned false (using the new <code>TokenStream</code> API).
3	abstract boolean <code>incrementToken()</code> Consumers (i.e., <code>IndexWriter</code>) use this method to advance the stream to the next token.
4	void <code>reset()</code> Resets this stream to the beginning.

Methods Inherited

This class inherits methods from the following classes:

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

Analyzer

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

Class Declaration

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

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

Class Constructors

The following table shows a class constructor:

S. No.	Constructor & Description
1	protected Analyzer()

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	void close() Frees persistent resources used by the Analyzer.
2	int getOffsetGap(Fieldable field) This is similar to getPositionIncrementGap(java.lang.String), except for Token offsets.
3	int getPositionIncrementGap(String fieldName) This is invoked before indexing a Fieldable instance if terms have already been added to that field.
4	protected Object getPreviousTokenStream() Used by Analyzers that implement reusable TokenStream to retrieve previously saved TokenStreams for re-use by the same thread.

5	TokenStream reusableTokenStream(String fieldName, Reader reader) Creates a TokenStream that is allowed to be re-used from the previous time that the same thread called this method.
6	protected void setPreviousTokenStream(Object obj) Used by Analyzers that implement reusableTokenStream to save a TokenStream for later re-use by the same thread.
7	abstract TokenStream tokenStream(String fieldName, Reader reader) Creates a TokenStream which tokenizes all the text in the provided Reader.

Methods Inherited

This class inherits methods from the following classes:

- java.lang.Object

This analyzer splits the text in a document based on the whitespace.

WhitespaceAnalyzer

This analyzer splits the text in a document based on whitespace.

Class Declaration

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

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

Class Constructors

S. No.	Constructor & Description
1	WhitespaceAnalyzer() Deprecated. Use WhitespaceAnalyzer(Version) instead.
2	WhitespaceAnalyzer(Version matchVersion) Creates a new WhitespaceAnalyzer.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	protected Reusable Analyzer Base. Token Stream Components create Components (String field Name, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

```
private void 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.CONTENTES,
            new StringReader(text));
    TermAttribute term = tokenStream.addAttribute(TermAttribute.class);
    while(tokenStream.incrementToken()) {
        System.out.print "[" + term.term() + " ] ";
    }
}
```

Example Application

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

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>Lucene - First</i>

	<i>Application</i> chapter as such for this chapter to understand the searching process.
2	Create <i>LuceneConstants.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep the rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure the 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 the 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 the creation of the source, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

```
[Lucene] [is] [simple] [yet] [powerful] [java] [based] [search] [library.]
```

SimpleAnalyzer

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

Class Declaration

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

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

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	SimpleAnalyzer() Deprecated. Use SimpleAnalyzer(Version) instead.
2	SimpleAnalyzer(Version matchVersion) Creates a new SimpleAnalyzer.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	protected Reusable Analyzer Base. Token Stream Components create Components (String field Name, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
```

```

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

```

LuceneTester.java

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

```

package com.tutorialspoint.lucene;

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

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

public class LuceneTester {

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

        tester = new LuceneTester();

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

    private void displayTokenUsingSimpleAnalyzer() throws IOException{

```

```

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

```

Running the Program

Once you are done with the creation of the source, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

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

StopAnalyzer

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

Class Declaration

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

```

public final class StopAnalyzer
    extends StopwordAnalyzerBase

```

Fields

Following are the fields for the **org.apache.lucene.analysis.StopAnalyzer** class:

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

Class Constructors

The following table shows the different class constructors:

S. No.	Constructor & Description
1	StopAnalyzer(Version matchVersion) Builds an analyzer which removes words in ENGLISH_STOP_WORDS_SET.
2	StopAnalyzer(Version matchVersion, File stopwordsFile) Builds an analyzer with the stop words from the given file.
3	StopAnalyzer(Version matchVersion, Reader stopwords) Builds an analyzer with the stop words from the given reader.
4	StopAnalyzer(Version matchVersion, Set<?> stopWords) Builds an analyzer with the stop words from the given set.

Class Methods

The following table shows the different class methods:

S. NO.	Method & Description
1	protected Reusable Analyzer Base. Token Stream Components create Components (String field Name, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents used to tokenize all the text in the provided Reader.

Methods Inherited

This class inherits methods from the following classes:

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

Usage

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

Example Application

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

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

```

package com.tutorialspoint.lucene;

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

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

public class LuceneTester {

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

        tester = new LuceneTester();

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

    private void displayTokenUsingStopAnalyzer() throws IOException{

```



```

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

```

Running the Program

Once you are done with the creation of the source, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

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

StandardAnalyzer

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

Class Declaration

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

```

public final class StandardAnalyzer
    extends StopwordAnalyzerBase

```

Fields

Following are the fields for the **org.apache.lucene.analysis.StandardAnalyzer** class:

- **static int DEFAULT_MAX_TOKEN_LENGTH** – This is the 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

The following table shows the different class constructors:

S. No.	Constructor & Description
1	StandardAnalyzer(Version matchVersion) Builds an analyzer with the default stop words (STOP_WORDS_SET).
2	StandardAnalyzer(Version matchVersion, File stopwords) Deprecated. Use StandardAnalyzer(Version, Reader) instead.
3	StandardAnalyzer(Version matchVersion, Reader stopwords) Builds an analyzer with the stop words from the given reader.
4	StandardAnalyzer(Version matchVersion, Set<?> stopWords) Builds an analyzer with the given stop words.

Class Methods

The following table shows the different class methods:

S. No.	Method & Description
1	protected Reusable Analyzer Base. Token Stream Components create Components(String fieldName, Reader reader) Creates a new ReusableAnalyzerBase.TokenStreamComponents instance for this analyzer.
2	int getMaxTokenLength()
3	void setMaxTokenLength(int length) Sets maximum allowed token length.

Methods Inherited

This class inherits methods from the following classes:

- org.apache.lucene.analysis.StopwordAnalyzerBase
- org.apache.lucene.analysis.ReusableAnalyzerBase
- org.apache.lucene.analysis.Analyzer

- java.lang.Object

Usage

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

Example Application

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

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

LuceneConstants.java

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

```
package com.tutorialspoint.lucene;

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

LuceneTester.java

This class is used to test the searching capability of the 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 the creation of the source, you can proceed by compiling and running your program. To do this, keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it will print the following message in Eclipse IDE's console:

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

11. 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 the 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 sorting mode is used by Lucene. Here, the 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 the sorting process.

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

2	Create <i>LuceneConstants.java</i> and <i>Searcher.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep the rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure the 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 the 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.CONTENTES,
            new StandardAnalyzer(Version.LUCENE_36));
    }

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

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

    public 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 the 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

We have used 10 text files from record1.txt to record10.txt containing 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 in the chapter **Lucene - Indexing Process**, you can see the list of index files created in that folder.

Running the Program

Once you are done with the creation of the source, the raw data, the data directory, the index directory and the indexes, you can compile and run your program. To do this, Keep the **LuceneTester.Java** file tab active and use either the **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If your application runs successfully, it 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
```