APACHE LUCENE

search engine programming

tutorialspoint

SIMPLYEASYLEARNING

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

Table of Contents

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.

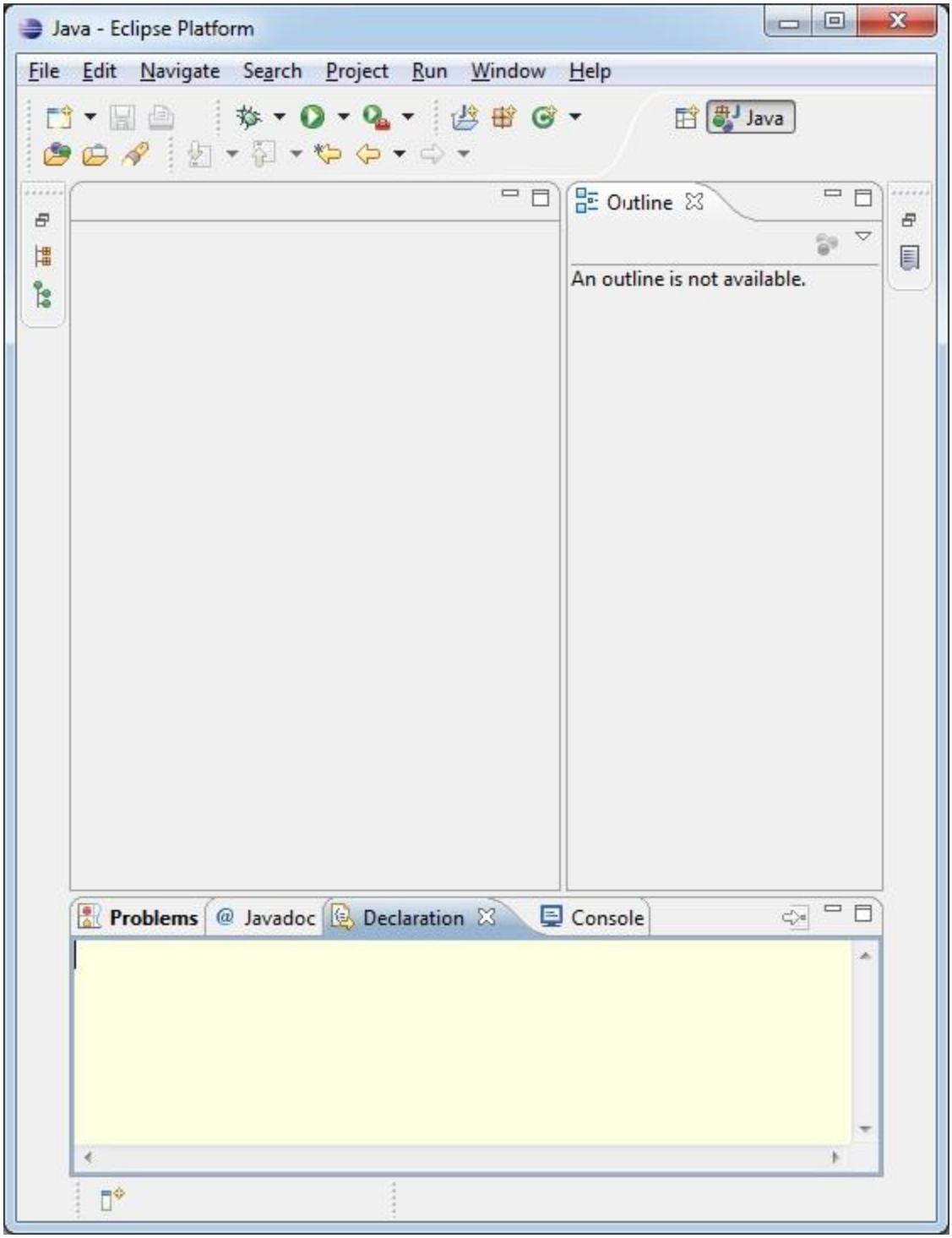
Eclipse can be started by executing the following commands on windows machine, or you can simply double click on **eclipse.exe**

```
%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

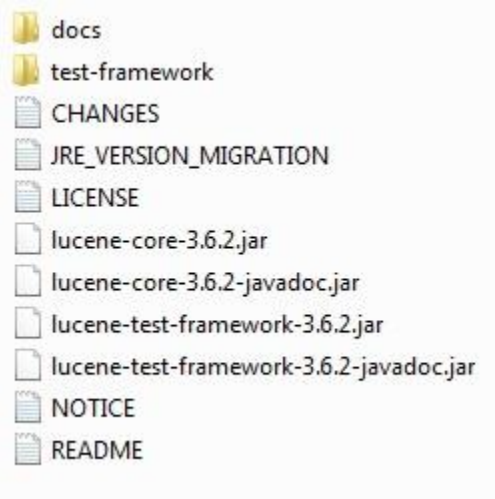
If the startup is successful, then you can proceed to set up your Lucene framework. Following are the simple steps to download and install the framework on your machine.

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 .tz file for Unix.

- Download the suitable version of Lucene framework binaries from http://archive.apache.org/dist/lucene/java/.

- At the time of writing this tutorial, I downloaded lucene-3.6.2.zip on my Windows machine and when you unzip the downloaded file it will give you 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.

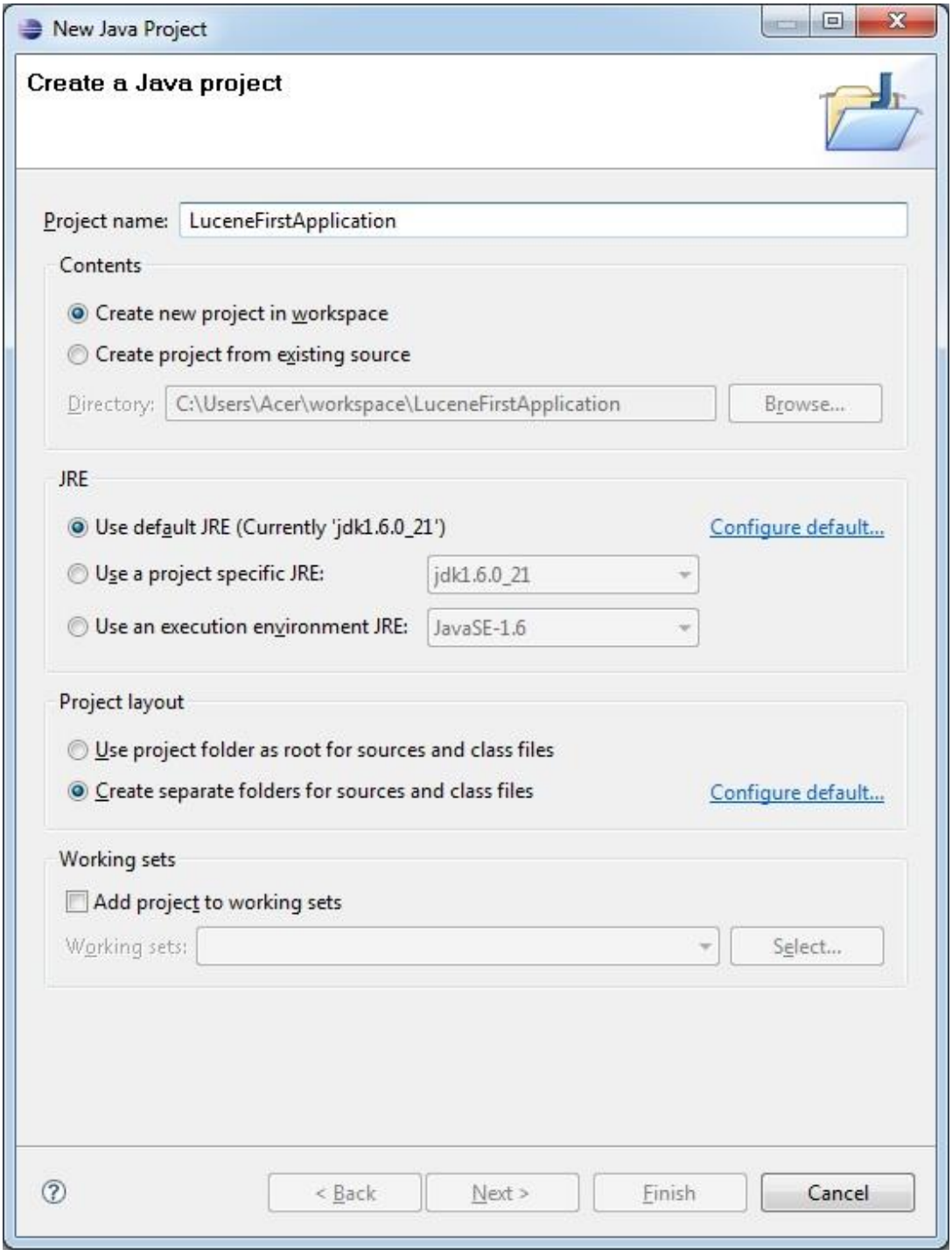
3. LUCENE – FIRST APPLICATION

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.

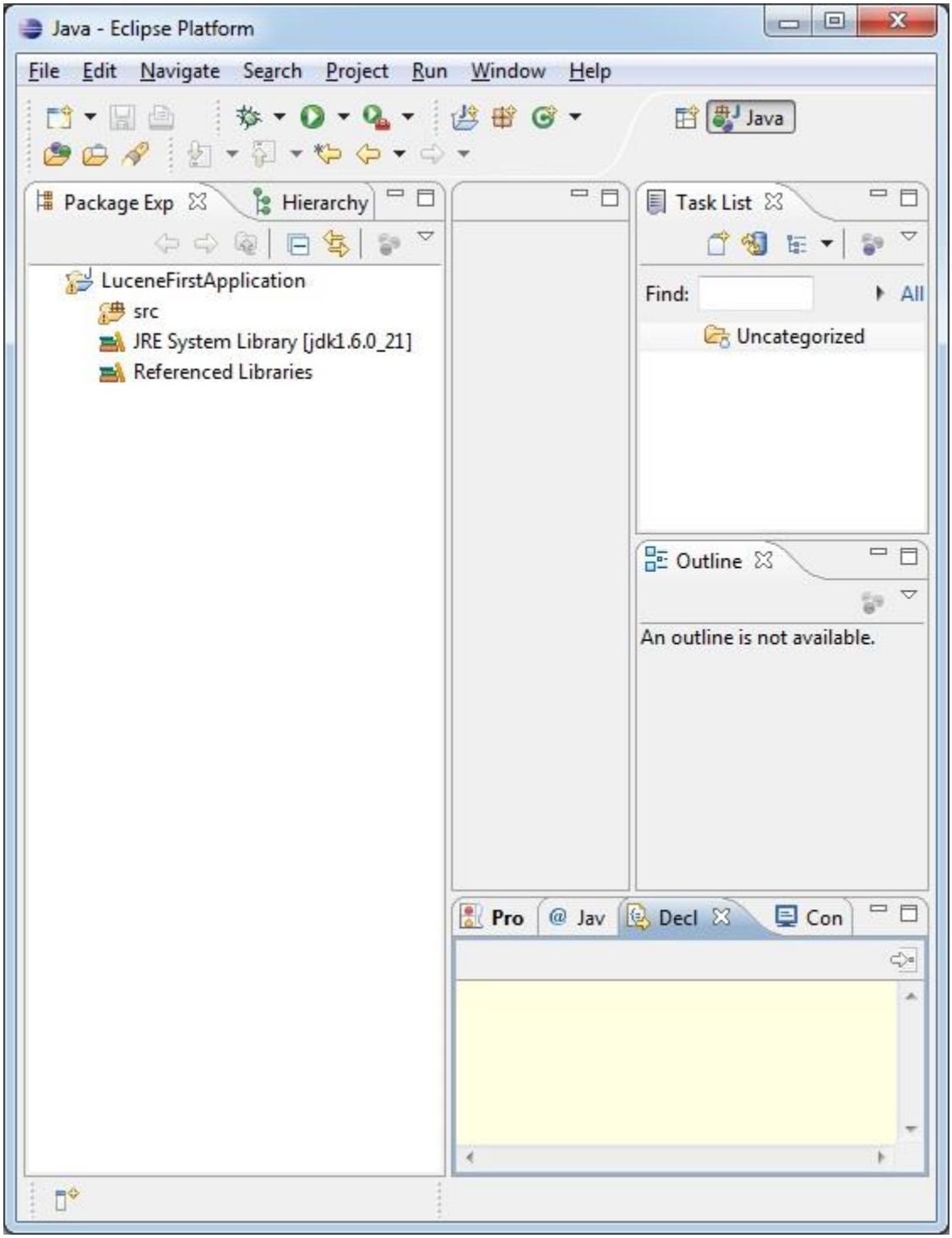
Let us now proceed by writing a simple Search Application which will print the number of search results found. We'll also see the list of indexes created during this process.

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:

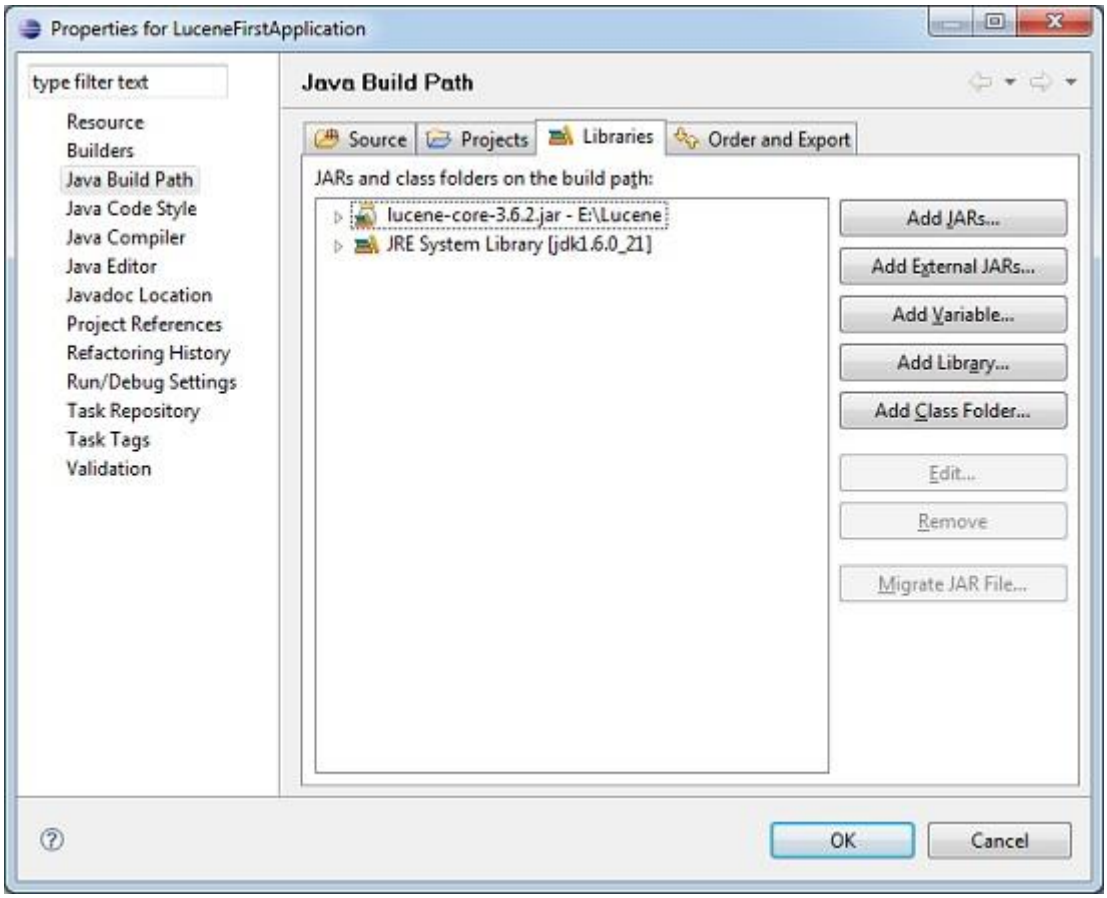


Once your project is created successfully, you will have following content in your **Project Explorer:**



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

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

   return document;
}

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

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

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

Searcher.java

This class is used to search the indexes created by 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

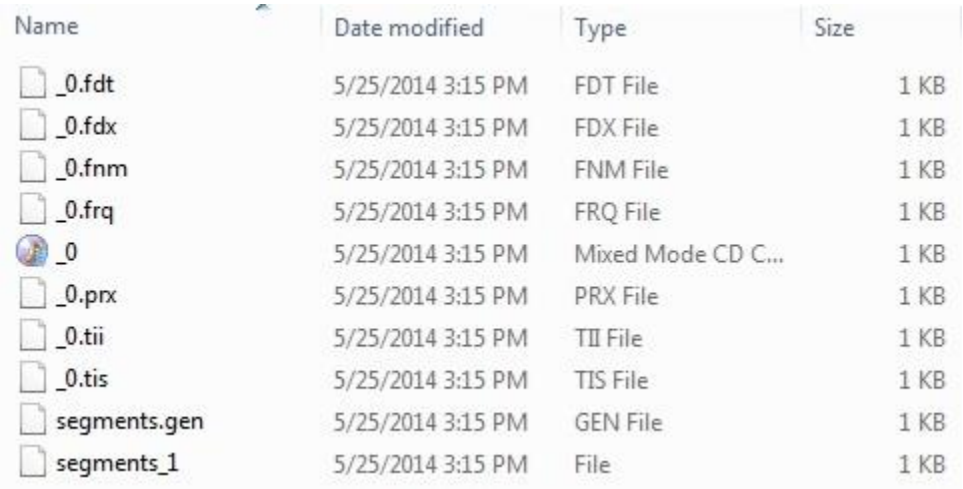
We have used 10 text files from record1.txt to record10.txt containing names and other details of the students and put them in the directory **E:\Lucene\Data**. Test Data. An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files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

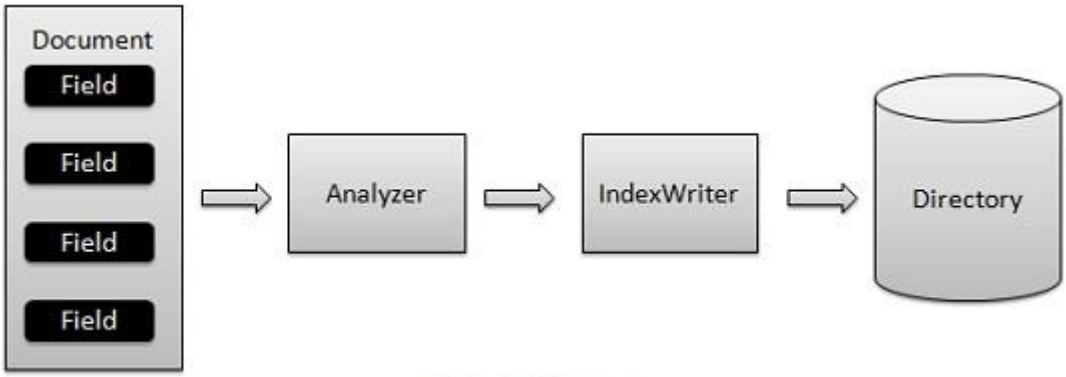
Once you've run the program successfully, you will have the following content in your **index directory:**



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	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

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.



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	**IndexWriter(Directory d, Analyzer a, boolean create, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl)** Deprecated. Use IndexWriter(Directory, IndexWriterConfig) instead.
2	**IndexWriter(Directory d, Analyzer a, boolean create, IndexWriter.MaxFieldLength mfl)** Deprecated. Use IndexWriter(Directory, IndexWriterConfig) instead.
3	**IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl)** Deprecated. Use IndexWriter(Directory, IndexWriterConfig) instead.
4	**IndexWriter(Directory d, Analyzer a, IndexDeletionPolicy deletionPolicy, IndexWriter.MaxFieldLength mfl, IndexCommit commit)** Deprecated. Use IndexWriter(Directory, IndexWriterConfig) instead.

5	**IndexWriter(Directory d, Analyzer a, IndexWriter.MaxFieldLength mfl)** Deprecated. Use IndexWriter(Directory, IndexWriterConfig) instead.
6	**IndexWriter(Directory d, IndexWriterConfig conf)** Constructs a new IndexWriter per the settings given in conf.

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 pcp)** Sets the PayloadProcessorProvider to use when merging payloads.
86	**void setRAMBufferSizeMB(double mb)** Deprecated. Use IndexWriterConfig.setRAMBufferSizeMB(double) instead.
87	**void setReaderTermsIndexDivisor(int divisor)** Deprecated. Use IndexWriterConfig.setReaderTermsIndexDivisor(int) instead.
88	**void setSimilarity(Similarity similarity)** Deprecated. Use IndexWriterConfig.setSimilarity(Similarity) instead.
89	**void setTermIndexInterval(int interval)** Deprecated. Use IndexWriterConfig.setTermIndexInterval(int).
90	**void setUseCompoundFile(boolean value)** Deprecated. Use LogMergePolicy.setUseCompoundFile(boolean).
91	**void setWriteLockTimeout(long writeLockTimeout)** Deprecated. Use IndexWriterConfig.setWriteLockTimeout(long) instead.
92	**static void unlock(Directory directory)** Forcibly unlocks the index in the named directory.
93	**void updateDocument(Term term, Document doc)** Updates a document by first deleting the document(s) containing term and then adding the new document.
94	**void updateDocument(Term term, Document doc, Analyzer analyzer)** Updates a document by first deleting the document(s) containing term and then adding the new document.

95	**void updateDocuments(Term delTerm, Collection<Document> docs)** Atomically deletes documents matching the provided **delTerm** and adds a block of documents with sequentially assigned document IDs, such that an external reader will see all or none of the documents.
96	**void updateDocuments(Term delTerm, Collection<Document> docs, Analyzer analyzer)** Atomically deletes documents matching the provided delTerm and adds a block of documents, analyzed using the provided analyzer, with sequentially assigned document IDs, such that an external reader will see all or none of the documents.